Java™ magazine

By and for the Java community

NETBEANS' FATE 03 | DESIGN PATTERNS IN JAVA 8 55 | CLOUD PROGRAMMING 63

JUnit 5 Arrives!

14 WHAT'S NEW IN JUNIT 5?

20 RUNNING TESTS FROM JUNIT 4 AND 5

25 JUNIT 5 EXTENSIONS

36 KENT BECK ON HIS EVOLVING VIEWS OF TESTING

43 MUTATION TESTING

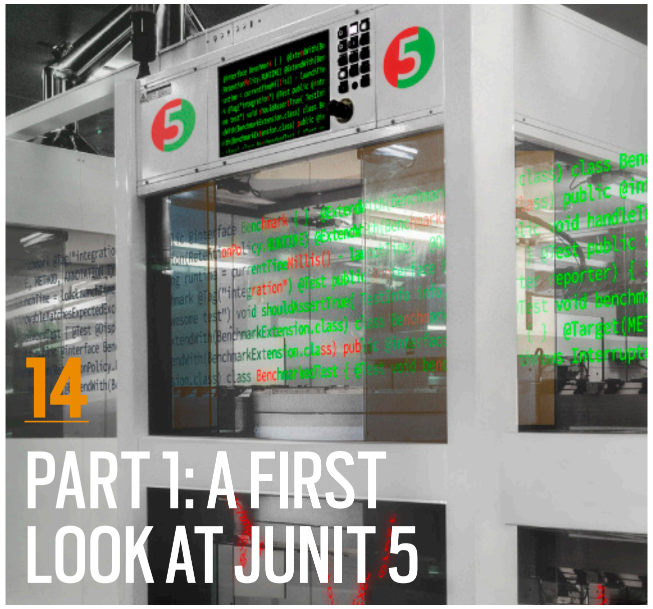
SPECIAL ISSUE

NOVEMBER/DECEMBER 2016

ORACLE.COM/JAVAMAGAZINE

ORACLE®

//table of contents /



COVER ART BY I-HUA CHEN

EDITORIAL

Editor in Chief
Andrew Binstock

Managing Editor
Claire Breen

Copy Editors
Karen Perkins, Jim Donahue

Technical Reviewer
Stephen Chin

DESIGN

Senior Creative Director
Francisco G Delgadillo

Design Director
Richard Merchán

Senior Designer
Arianna Pucherelli

Designer
Jaime Ferrand

Senior Production Manager
Sheila Brennan

Production Designer
Kathy Cygnarowicz

PUBLISHING

Publisher
Jennifer Hamilton +1.650.506.3794

Associate Publisher and Audience Development Director
Karin Kinnear +1.650.506.1985

Audience Development Manager
Jennifer Kurtz

ADVERTISING SALES

Sales Director
Tom Cometa

Account Manager
Mark Makinney

Account Manager
Marcin Gamza

Advertising Sales Assistant
Cindy Elhaj +1.626.396.9400 x 201

Mailing-List Rentals
Contact your sales representative.

RESOURCES

Oracle Products
+1.800.367.8674 (US/Canada)

Oracle Services
+1.888.283.0591 (US)

ARTICLE SUBMISSION

If you are interested in submitting an article, please email the editors.

SUBSCRIPTION INFORMATION

Subscriptions are complimentary for qualified individuals who complete the subscription form.

MAGAZINE CUSTOMER SERVICE

java@halldata.com **Phone** +1.847.763.9635

PRIVACY

Oracle Publishing allows sharing of its mailing list with selected third parties. If you prefer that your mailing address or email address not be included in this program, contact Customer Service.

Java Magazine is published bimonthly and made available at no cost to qualified subscribers by
Oracle, 500 Oracle Parkway, MS OPL-3A, Redwood City, CA 94065-1600.

NetBeans Gets a New Life—or Does It?

The transition from Oracle to the Apache Software Foundation marks the beginning of an uncertain new era for the Java IDE.

At JavaOne this year, the NetBeans community announced that the project was moving from its longtime home at Oracle to the Apache Software Foundation (ASF). In a history that dates back some 20 years, this will be NetBeans' fifth new home, showing the product's remarkable power of endurance. An important question is whether working under the aegis of the ASF will bring NetBeans new life and new aficionados, or whether it signals the final chapter of a storied lifeline.

As many readers know, NetBeans is one of the four principal Java IDEs. The others are the open source Eclipse from the Eclipse Foundation, IntelliJ IDEA from JetBrains (consisting of an open source version and a higher-end closed source version), and JDeveloper (a free, closed source IDE from Oracle). What few readers might know is that NetBeans was the first of these products—beating Borland's JBuilder by a year. (JDeveloper, which was based on JBuilder, was next, followed years later by Eclipse and IntelliJ.)

NetBeans became a popular Java IDE because of several features, most especially the lightness of its use. While competing products had a long setup cycle for new projects and a comparatively "heavy" feel, NetBeans was great for coding on the fly and always felt light and responsive. While it lacked some of its competitors' code-management features, it was the first to offer a built-in execution profiler and, if I recall correctly, the only one to include a small-scale J2EE server, OC4J, to quickly test web projects locally. It was also the first IDE to offer a top-quality Swing-

PHOTOGRAPH BY BOB ADLER/GETTY IMAGES

based GUI-development tool, called Matisse.

That's a lot of quality to come from what was originally a student project at Charles University in Prague. (The core development team for NetBeans has remained primarily based in Prague, although marketing and other functions have been based at various times in the United States and elsewhere.)

Eventually, NetBeans was acquired by Sun, where it was open sourced. And through the 2011 acquisition of Sun, NetBeans became part of Oracle. At that point, I was quite surprised to read of Oracle's commitment to continue developing NetBeans. After all, the company already offered JDeveloper for free and sponsored Oracle-specific packages and extensions for Eclipse. But actually, Oracle did more than just commit to supporting the platform's development and promotion; it also began using portions of NetBeans in its own products, specifically JDeveloper and VisualVM, and eventually a variety of other development tools. For this reason, even with the move to the ASF, NetBeans has secured a commitment from Oracle to underwrite its development for two more releases: the upcoming 8.x version and the 9.0 release.

If you were to view NetBeans purely as a programming environment, its fate after Oracle's commitment expires would be most uncertain. Although many projects under the ASF aegis have flourished (Maven, Hadoop, Spark, and others), more than a few projects have migrated to the ASF only to die there. (See the Apache Attic for a list of defunct projects.) However, over the years, NetBeans evolved from an IDE into a platform consisting of large-scale components that can be assembled in different ways to form desktop applications. This architecture uses a rather different approach than Eclipse's OSGi-based system of modules and bundles. (This page compares the Eclipse and NetBeans architectures.) Numerous companies—including Oracle—have exploited the benefits of NetBeans' architecture and built applications whose runtime includes the platform components.

These companies have an interest in continuing the forward direction of NetBeans, and some have committed to work on NetBeans in its new home. I expect—but obviously I don't know—that they will contribute either directly or by engaging NetBeans' current cohort of developers to continue developing the platform. In addition, the community of users, many of whom are truly dedicated to NetBeans, might well step up and begin contributing. It's difficult to project the extent of participation because very few projects with so large a user base have been migrated to the ASF, and so there is little history to provide guidance.

For users of NetBeans, though, nothing need be done for now or in the near term. The 9.0 release is scheduled for August 2017 and will cover Java 9. By that time, we will surely have more insight into the transition of NetBeans, the level of activity, and the level of support from both commercial users and the developer community.

Andrew Binstock, Editor in Chief
javamag_us@oracle.com
@platypusguy

JRebel
Java just got faster...

Java magazine
By and for the Java community
JAX-RS 39 | JAVA INTERFACES 47 | FANTOM 53
SEPTEMBER/OCTOBER 2016
Devices and IoT
14 INTERACTING WITH SENSORS
24 CONTROLLING CNC ROUTING FROM A RASPBERRY PI
32 IOT AND THE CLOUD
ORACLE.COM/JAVAMAGAZINE
ORACLE

SEPTEMBER/OCTOBER 2016

Limited Language Choices

I enjoyed the *Java Magazine* article "Appreciating Limited Choice in Languages" (September/October 2016), which discusses the benefits that Java enjoys over other languages (such as C) in having a reasonable amount of standardization. However, there's a problem: Oracle's recommended style guidelines for Java are hopelessly out of date—and have been for years.

One huge indicator of this problem: The manual on Java language style (which should be the bible here) has not been updated since 1999.

The article says, "The convenience and benefits of such strictures that ensure uniform syntax are widely recognized." I agree. So, when will Oracle update the Java language style manual and offer guidance to developers and organizations?

—Alan Zeichick
Phoenix, AZ

Andrew Binstock responds: "I contacted the Java team to get more detail on this. They told me that the document you're referring to was not posted as an attempt to codify the language and therefore be a regularly updated document. Rather, it was a set of internal coding guidelines published in response to a community request. The team suggested that the most normative examples of Java style are those used in Oracle's Java Tutorials.

"Most books available today on Java style were written a long time ago, alas. A volume from 2014 that's a worthy reference is Java Coding Guidelines*, by Fred Long and a group of security experts. It not so much defines a style as provides coding guidelines for writing secure code. For my own use, when I need a full set of well-reasoned Java coding guidelines, I generally turn to the* Google Java Style Guide*."*

Regarding your editorial, Python dominates UNIX installation and infrastructure. Talk about a language needing "prescriptive" control!

—Richard Elkins
Dallas, TX

Back Issues Prior to 2015

I was reading something on the web that mentioned an article in an older version of *Java Magazine*. However, when I look on the magazine home page it does not show any issues before 2015. Is there any way to access the back issues? Even if they're just in PDF format, that would be helpful.

—Michael Szela
Palatine, IL

Andrew Binstock responds: "We recently made the 2015 and 2016 back issues available for viewing on the web and downloading as PDF files. We plan to make the previous issues of Java Magazine *available soon, most likely as downloadable PDF files for subscribers. We'll announce the availability of each year in the letter that goes out to subscribers with each new issue of the magazine. We gratefully appreciate your patience."*

Contact Us

We welcome your comments and suggestions. Write to us at javamag_us@oracle.com. For other ways to reach us, see the last page of this issue.

Jfokus *FEBRUARY 6, UNIVERSITY DAY; FEBRUARY 7–8, CONFERENCE*
STOCKHOLM, SWEDEN
Jfokus is the largest annual Java developer conference in Sweden. Conference topics include Java SE and Java EE, continuous delivery and DevOps, IoT, cloud and big data, trends, and JVM languages. This year, the first day of the event will include a VM Tech Summit, which is an open technical collaboration among language designers, compiler writers, tool builders, runtime engineers, and VM architects. The schedule will be divided equally between traditional presentations of 45 minutes and informal, facilitated deep-dive discussion groups among smaller, self-selected participants. Space is limited, as this summit is organized around a single classroom-style room to support direct communication between participants.

PHOTOGRAPH BY CHAS B/FLICKR

ArchConf
DECEMBER 12–15, 2016
CLEARWATER, FLORIDA
ArchConf is an educational event for software architects, technical leaders, and senior developers presented by the No Fluff Just Stuff software symposium. Among the slated sessions are Modularity, Microservices, and Modern Architectural Paradigms, led by Kirk Knoernschild, author of Prentice-Hall's *Java Application Architecture*, and Cloud-Native Application Architecture, led by Matt Stine, author of O'Reilly's *Migrating to Cloud-Native Application Architectures*.

DevConf.cz 2017
JANUARY 27–29
BRNO, CZECH REPUBLIC
DevConf.cz 2017 is a free three-day open-source Fedora Linux and JBoss community conference for Red Hat and community developers, DevOps professionals, testers, and documentation writers. Set to be hosted at the Brno University of Technology, all talks, presentations, and workshops will be conducted in English. Several tracks are usually devoted specifically to Java EE, and the conference can be attended online.

DevNexus
FEBRUARY 20–22
ATLANTA, GEORGIA
DevNexus is devoted to connecting developers from all over the world, providing affordable education, and promoting open source values. The 2017 conference will take place at the Georgia World Congress Center in downtown Atlanta. Presenters will include Josh Long, author of O'Reilly's upcoming *Cloud Native Java: Designing Resilient Systems with Spring Boot*, *Spring Cloud, and Cloud Foundry*, and Venkat Subramaniam, author of Pragmatic's *Functional Programming in Java: Harnessing the Power of Java 8 Lambda Expressions*.

Voxxed Days Zürich
FEBRUARY 23
ZÜRICH, SWITZERLAND
Sharing the Devoxx philosophy that content comes first, Voxxed Days events see both

internationally renowned and local speakers converge. Past presentations have included Bringing the Performance of Structs to Java (Sort Of) by Simon Ritter and Java Security Architecture Demystified by Martin Toshev.

Topconf Linz 2017

FEBRUARY 28, WORKSHOPS
MARCH 1–2, CONFERENCE
LINZ, AUSTRIA

Topconf covers Java and JVM, DevOps, reactive architecture, innovative languages, UX/UI, and agile development. Presentations this year include Java Libraries You Can't Afford to Miss, 8 Akka Antipatterns You'd Better Be Aware Of, and Spring Framework 5: Reactive Microservices on JDK 9.

QCon London 2017

MARCH 6–8, CONFERENCE
MARCH 9–10, WORKSHOPS
LONDON, ENGLAND

For more than a decade, QCon London has empowered software development by facilitating the spread of knowledge and innovation in the developer community. Scheduled tracks this year include Performance Mythbusting and Every Last Nanosecond: Low Latency Java.

jDays

MARCH 7–8
GOTHENBURG, SWEDEN

jDays brings together software engineers around the world to share their experiences in different areas such as Java, software engineering, IoT, digital trends, testing, agile methodologies, and security.

ConFoo Montreal 2017

MARCH 8–10
MONTREAL, QUEBEC, CANADA

ConFoo Montreal is a multitechnology conference for web developers that promises 155 presentations by popular international speakers. Past ConFoo topics have included how to write better streams with Java 8 and an introduction to Java 9.

Embedded World

MARCH 14–16
NUREMBERG, GERMANY

The theme for the 15th annual gathering of embedded system developers is Securely Connecting the Embedded World. Topics include IoT, connectivity, software engineering, and security.

Devoxx US

MARCH 21–23
SAN JOSE, CALIFORNIA

Devoxx US focuses on Java, web, mobile, and JVM languages. The conference includes more than 100 sessions in total, with tracks devoted to server-side Java, architecture and security, cloud and containers, big data, IoT, and more.

JavaLand

MARCH 28–30
BRÜHL, GERMANY

This annual conference features more than 100 lectures on subjects such as core Java and JVM languages, enterprise Java and cloud technologies, IoT, front-end and mobile computing, and much more. Scheduled presentations include Multiplexing and Server Push: HTTP/2 in Java 9, The Dark and Light Side of JavaFX, JDK 8 Lambdas: Cool Code that Doesn't Use Streams, Migrating to Java 9 Modules, and Java EE 8: Java EE Security API.

PHOTOGRAPH BY CRALVAREZ/FLICKR

O'Reilly Software Architecture Conference

APRIL 2–3, TRAINING
APRIL 3–5, TUTORIALS AND CONFERENCE
NEW YORK, NEW YORK

This event promises four days of in-depth professional training that covers software architecture fundamentals; real-world case studies; and the latest trends in technologies, frameworks, and techniques. Past presentations have included Introduction to Reactive Applications, Reactive Streams, and Options for the JVM, as well as Microservice Standardization.

Devoxx France

APRIL 5, WORKSHOPS
APRIL 6–7, CONFERENCE
PARIS, FRANCE

Devoxx France presents keynotes from prestigious speakers, then a cycle of eight mini conferences every 50 minutes. You can build your own calendar and follow the sessions as you wish. Founded by developers for developers, Devoxx France covers topics ranging from web security to cloud computing. (No English page available.)

J On The Beach

MAY 17 AND 20, WORKSHOPS
MAY 18–19, TALKS
MALAGA, SPAIN

JOTB is an international rendezvous for developers interested in big data technologies. JVM and .NET technologies, embedded and IoT development functional programming, and data visualization will all be discussed. Scheduled speakers include longtime Java Champion Martin Thompson and Red Hat Director of Developer Experience Edson Yanaga.

QCon New York

JUNE 26–28, CONFERENCE
JUNE 29–30, WORKSHOPS
NEW YORK, NEW YORK

QCon is a practitioner-driven conference for technical team leads, architects, engineering directors, and project managers who influence innovation in their teams. Past speakers include Chris Richardson, author of *POJOs in Action*, and Frank Greco, organizer of the largest Java user group in North America (NYJavaSIG).

JCrete

JULY 17–21
KOLYMBARI, GREECE

This loosely structured "unconference" involves morning sessions discussing all things Java, combined with afternoons spent socializing, touring, and enjoying the local scene. There is also a JCrete4Kids component for introducing youngsters to programming and Java. Attendees often bring their families.

ÜberConf

JULY 18–21
DENVER, COLORADO

ÜberConf 2017 will be held at the Westin Westminster in downtown Denver. Topics include Java 8, microservice architectures, Docker, cloud, security, Scala, Groovy, Spring, Android, iOS, NoSQL, and much more.

NFJS Boston

SEPTEMBER 29–OCTOBER 1
BOSTON, MASSACHUSETTS

Since 2001, the No Fluff Just Stuff (NFJS) Software Symposium Tour has delivered more than 450 events with more than 70,000 attendees. This event in Boston covers the latest trends within the Java and JVM ecosystem, DevOps, and agile development environments.

JavaOne

OCTOBER 1–5
SAN FRANCISCO, CALIFORNIA

Whether you are a seasoned coder or a new Java programmer, JavaOne is the ultimate source of technical information and learning about Java. For five days, Java developers gather from around the world to talk about upcoming releases of Java SE, Java EE, and JavaFX; JVM languages; new development tools; insights into recent trends in programming; and tutorials on numerous related Java and JVM topics.

Have an upcoming conference you'd like to add to our listing? Send us a link and a description of your event four months in advance at javamag_us@oracle.com.

DEVOXX™ IS COMING TO THE UNITED STATES

DEVOXX US 2017
SAN JOSE CONVENTION CENTER
MARCH 21-23, 2017

DEVOXX US FACTS

OVER 200 TECHNICAL SESSIONS

TARGETING 1000+ DEVELOPERS

AD HOC SPONSORSHIP OPPORTUNITIES GIVING YOU UNIQUE REACH TO DEVELOPERS

VARIETY OF SPONSORSHIP PACKAGES, STARTING AT $5,000.

WANT TO BECOME A SPONSOR?

THE DEVOXX WAY - KEY DIFFERENTIATORS

Devoxx is the largest developer community conference series on the planet. Devoxx events are held annually in Belgium, France, UK, Poland, Morocco, and ... beginning in 2017, in the USA.

Our mission is to offer passionate developers the best place to network, hack, get inspired and learn.

- All conference sessions are recorded and published online
- Curated by developers, for developers
- Devoxx is vendor neutral and community driven
- Low cost entrance fee attracts developers from around the globe

SPONSORSHIP INFORMATION

CONTACT SPONSORS@DEVOXX.US OR VISIT DEVOXX.US/SPONSORS-2017

JavaOne 2016: The News and the Videos

JavaOne, the annual meeting of Java developers, was held this year as usual in San Francisco, California, in mid-September. This conference has long been the largest congregation of Java programmers in the world. This year, more than 9,000 developers participated in hundreds of sessions, many of which dealt with new technologies such as the upcoming JDK 9.

There is news to report that came out of the conference. Perhaps most important is Oracle's announcement of the forthcoming Java EE 8, which will be followed by Java EE 9. Oracle Group Vice President Anil Gaur gave some details on version 8, which is expected to ship toward the end of 2017. It will include enhanced security (management of secrets and support for OAuth and OpenID), greater support for reactive programming, a unified event model, HTTP/2, JSON-B, as well as a large set of other technologies. In addition, it will be optimized for serverless contexts and for containers. Java EE 9 is expected to ship a year later, in late 2018. (A video of Gaur's presentation on Java EE 8 is available on YouTube.)

Another event of importance that happened during JavaOne, but not strictly speaking *at* the conference, was the changing fate of the NetBeans IDE. Oracle announced that it would be the primary sponsor of the IDE for the next two releases only—that is, the imminent 8.x release and the later version 9, which will support Java 9. In response, the NetBeans community has applied to become a project under the aegis of the Apache Software Foundation. The ramifications of this are discussed in detail in the editorial in this issue (page 3).

As usual, there were many brilliant presentations at JavaOne. Some 85 of them were videotaped and are now posted on YouTube.

An annual fixture of JavaOne is the Duke's Choice Awards, which recognize particularly meritorious Java projects and Java tools. These awards are chosen by a panel of community leaders and Oracle technical staff. This year, the judges selected the following recipients of interest to developers: PrimeFaces (there's a pattern here—last year the judges chose OmniFaces); HeapStats, a lightweight monitoring tool for Java heap and GC status with a GUI front end for analyzing the data it generates; and Strata, an open source package used for financial analytics and market risk estimation.

The next annual JavaOne conference will be held October 1–5, 2017, in San Francisco. For a listing of other conferences and events, see the Events section (page 7) in this issue.

SPECIAL ISSUE

JUnit 5: The New Generation of Unit Testing

JUnit is the most widely used testing tool in Java. Survey after survey shows this. It's so pervasive that other testing tools are frequently built on top of JUnit (Spock, for example, as well as most of the behavior-driven development frameworks), rather than trying to duplicate its capabilities. JUnit's speed, ease of use, and universality make it the Java developer's universal tool.

Version 5 is a fundamental rewrite and rearchitecture of JUnit. The new features of this release are summarized in our first article (page 14), which gives a concise overview of the refinements. The second article (page 20) shows how to include JUnit 5 in your toolchain and, especially, how to run tests for versions 4 and 5 in the same testing run.

Now that you see the benefits of this release, we take you for a deep dive into the architecture (page 25). This article is ideal for developers who want to extract the most capability from the new version, rather than just stay at a basic assertion level. It's also an excellent introduction to JUnit's extension points, which implement the design that tools use to drive or interact with JUnit.

But this is a special issue, so there's plenty more here. On page 36, we interview Kent Beck, the original parent of JUnit as well as the father of extreme programming—the core set of practices that form the basis of most modern software development. As you'll see, Beck's view on testing has evolved from being deeply rooted in test-first development to one that balances the benefits of tests with the costs they impose. He explains this more nuanced view in detail, which is sure to give pause to die-hard TDD fans.

Finally, for developers who rely on unit testing as a backstop for mission-critical code, we explore mutation testing (page 43), which is heavy-duty, automated testing that searches for gaps in unit tests. It takes the unit tests and modifies them slightly to see if tested-for conditions when changed or removed from the test cause the test to fail or throw an exception. This can identify duplicate tests, incomplete tests, and those that don't actually test what you expect.

May the green bar be good to you!

ART BY I-HUA CHEN

MERT ÇALIŞKAN

Part 1: A First Look at JUnit 5

The long-awaited release of JUnit 5 is a complete redesign with many useful additions.

JUnit, the widely used Java unit testing framework, has just seen the first alpha release of version 5 after 10 years on version 4. JUnit 5 consists of a revamped codebase with a modular architecture, a new set of annotations, an extensible model for third-party library integration, and the ability to use lambda expressions in assertions.

The predecessor of JUnit 5 was the JUnit Lambda project, which sowed the first ideas for the next generation of unit testing and was crowd-funded until October 2015 on Indiegogo, where it received more than twice the target amount in contributions.

Through the years, JUnit has captured the essence of what a unit testing framework should be. However, its core mostly stayed intact, which made it difficult for it to evolve. This new version is a complete rewrite of the whole product that aims to provide a sufficient and stable API for running and reporting tests. Implementing unit tests with JUnit 5 requires Java 8 at a minimum, but it can run tests on code written for earlier versions of Java.

In Part 1 of this article, I describe the principal new features of JUnit 5, illustrating them with detailed examples. The JUnit team is planning to ship the final version of the framework by the end of 2016. Milestone 2 is one of the last steps before JUnit 5 officially ships. This will surely be one of the most consequential releases ever in the Java ecosystem. In Part 2 (page 20), I explain how to use and configure JUnit 5 with your existing tools and how to run tests from JUnit 4 and JUnit 5 together.

Anatomy of a JUnit 5 Test

Let's look at some JUnit 5 tests, starting with the simple JUnit test shown in **Listing 1**.

Listing 1.

```
import org.junit.jupiter.api.*;

class SimpleTest {

    @Test
    void simpleTestIsPassing() {
        org.junit.jupiter.api.Assertions
           .assertTrue(true);
    }
}
```

For a simple JUnit 5 test class, such as the one shown here, there is almost no difference to be seen at first glance when compared with a JUnit 4 test class. The main difference is that there is no need to have test classes and methods defined with the `public` modifier. Also, the `@Test` annotation—along with the rest of the annotations—has moved to a new package named `org.junit.jupiter.api`, which needs to be imported.

Capitalizing on the Power of Annotations

JUnit 5 offers a revised set of annotations, which, in my view, provide essential features for implementing tests. The annotations can be declared individually, or they can be composed to create custom annotations. In the following section, I describe each annotation and give details with examples.

@DisplayName. It's now possible to display a name for a test class or its methods by using the @DisplayName annotation. As shown in Listing 2, the description can contain spaces and special characters. It can even contain emojis, such as ☺.

Listing 2.

```
@DisplayName("This is my awesome test class ‰")
class SimpleNamedTest {

    @DisplayName("This is my lonely test method")
    @Test
    void simpleTestIsPassing() {
        assertTrue(true);
    }
}
```

@Disabled. The @Disabled annotation is analogous to the @Ignore annotation of JUnit 4, and it can be used to disable the whole test class or one of its methods from execution. The reason for disabling the test can be added as a description to the annotation, as shown in Listing 3.

Listing 3.

```
class DisabledTest {

    @Test
    @Disabled("test is skipped")
    void skippedTest() {
        fail("feature not implemented yet");
    }
}
```

@Tags and @Tag. It's possible to tag test classes, their methods, or both. Tagging provides a way of filtering tests for execution. This approach is analogous to JUnit 4's Categories. Listing 4 shows a sample test class that uses tags.

Listing 4.

```
@Tag("marvelous-test")
@Tags({@Tag("fantastic-test"), @Tag("awesome-test")})
class TagTest {

    @Test
    void normalTest() {
    }

    @Test
    @Tag("fast-test")
    void fastTest() {
    }
}
```

You can filter tests for execution or exclusion by providing tag names to the test runners. The way to run ConsoleLauncher is described in detail in Part 2 of this article. With ConsoleLauncher, you can use the -t parameter for providing required tag names or the -T parameter for excluding tag names.

@BeforeAll, @BeforeEach, @AfterEach, and @AfterAll. The behavior of these annotations is exactly the same as the behavior of JUnit 4's @BeforeClass, @Before, @After, and @AfterClass, respectively. The method annotated with

If an assumption fails, it does not mean the code is broken, but only that the test provides no useful information. The default JUnit runner ignores such failing tests. This approach enables other tests in the series to be executed.

@BeforeEach will be executed before each @Test method, and the method annotated with @AfterEach will be executed after each @Test method. The methods annotated with @BeforeAll and @AfterAll will be executed before and after the execution of all @Test methods. These four annotations are applied to the @Test methods of the class in which they reside, and they will also be applied to the class hierarchy if any exists. (Test hierarchies are discussed next.) The methods annotated with @BeforeAll and @AfterAll need to be defined as static.

@Nested test hierarchies. JUnit 5 supports creating hierarchies of test classes by nesting them inside each other. This option enables you to group tests logically and have them under the same parent, which facilitates applying the same initialization methods for each test. Listing 5 shows an example of using test hierarchies.

Listing 5.

```
class NestedTest {

    private Queue<String> items;

    @BeforeEach
    void setup() {
        items = new LinkedList<>();
    }

    @Test
    void isEmpty() {
        assertTrue(items.isEmpty());
    }

    @Nested
    class WhenEmpty {
      @Test
      public void removeShouldThrowException() {
        expectThrows(
          NoSuchElementException.class,
          items::remove);
      }
    }

    @Nested
    class WhenWithOneElement {
      @Test
      void addingOneElementShouldIncreaseSize() {
            items.add("Item");
            assertEquals(items.size(), 1);
      }
    }
}
```

Assertions and Assumptions

The org.junit.jupiter.api.Assertions class of JUnit 5 contains static assertion methods—such as assertEquals, assertTrue, assertNull, and assertSame—and their corresponding negative versions for handling the conditions in test methods. JUnit 5 leverages the use of lambda expressions with these assertion methods by providing overloaded versions that take an instance of java.util.function.Supplier. This enables the evaluation of the assertion message lazily, meaning that potentially complex calculations are delayed until a failed assertion. Listing 6 shows using a lambda expression in an assertion.

Listing 6.

```
class AssertionsTest {

    @Test
    void assertionShouldBeTrue() {
        assertEquals(2 == 2, true);
    }
    @Test
    void assertionShouldBeTrueWithLambda() {
        assertEquals(3 == 2, true,
               () -> “3 not equal to 2!”);
    }
}
```

The org.junit.jupiter.api.Assumptions class provides assumeTrue, assumeFalse, and assumingThat static methods. As stated in the documentation, these methods are useful for stating assumptions about the conditions in which a test is meaningful. If an assumption fails, it does not mean the code is broken, but only that the test provides no useful information. The default JUnit runner ignores such failing tests. This approach enables other tests in the series to be executed.

Grouping Assertions

It's also possible to group a list of assertions together. Using the assertAll static method, shown in **Listing 7**, causes all assertions to be executed together and all failures to be reported together.

Listing 7.

```
class GroupedAssertionsTest {

    @Test
    void groupedAssertionsAreValid() {
        assertAll(
            () -> assertTrue(true),
            () -> assertFalse(false)
        );
    }
}
```

Expecting the Unexpected

JUnit 4 provides a way to handle exceptions by declaring them as an attribute to the @Test annotation. This is an enhancement compared with previous versions that required the use of try-catch blocks for handling exceptions. JUnit 5 introduces the usage of lambda expressions for defining the exception inside the assertion statement. **Listing 8** shows the placement of the exception directly into the assertion.

Listing 8.

```
class ExceptionsTest {

    @Test
    void expectingArithmeticException() {
        assertThrows(ArithmeticException.class,
                     () -> divideByZero());
    }

    int divideByZero() {
        return 3/0;
    }
}
```

With JUnit 5, it's also possible to assign the exception to a variable in order to assert conditions on its values, as shown in **Listing 9**.

Listing 9.

```
class Exceptions2Test {

  @Test
  void expectingArithmeticException() {
    StringIndexOutOfBoundsException exception =
      expectThrows(
        StringIndexOutOfBoundsException.class,
        () -> "JUnit5 Rocks!".substring(-1));

    assertEquals(exception.getMessage(),
       "String index out of range: -1");
    }
}
```

Dynamic Testing

With a new dynamic testing feature of JUnit 5, it's now possible to create tests at runtime. This was not possible prior to version 5 because all testing code needed to be defined at compile time. An example of dynamic test creation is shown in **Listing 10**.

Listing 10.

```
class DynamicTestingTest {

    @TestFactory
    List<DynamicTest>
    createDynamicTestsReturnAsCollection() {
        return Arrays.asList(
              dynamicTest("A dynamic test",
              () -> assertTrue(true)),
              dynamicTest("Another dynamic test",
              () -> assertEquals(6, 3 * 2))
        );
    }
}
```

To create dynamic tests, first I created a method inside a class and annotated it with `@TestFactory`. JUnit handles this `@TestFactory` method while analyzing the class, and it dynamically creates testing units by using its return value. The method annotated with `@TestFactory` must return an instance of `Collection`, `Stream`, `Iterable`, or an `Iterator` of type `DynamicTest`. The `DynamicTest` class denotes a test case that will be generated at runtime. Actually, it's a wrapper class that contains a name and an executable. That executable refers to the test code execution block. The `dynamicTest` static method definition resides under the `DynamicTest` class, and its objective is to create an instance of `DynamicTest` by retrieving a name and an instance of `Executable`, which consists of lambda expressions (as shown in **Listing 10**) that use two assertions.

The JUnit team has succeeded in offering a new, redesigned version of JUnit that addresses nearly all the limitations of previous versions.

The lifecycle of a dynamic test is different from that of a standard `@Test` annotated method. This means that lifecycle callback methods, such as `@BeforeEach` and `@AfterEach`, are not executed for dynamic tests.

Parameterized Test Methods

With the help of dynamic testing in JUnit 5, it's possible to execute the same test with different data sets. This was also possible in JUnit 4 by employing the `Parameterized` runner and by defining data with the `@Parameterized.Parameters` annotation. But that approach has a limitation: it runs all test methods annotated with `@Test` for every parameter again and again, leading to needless executions. Creating dynamic tests for each data item could lead to better encapsulation that is local to the test method. I demonstrate this in **Listing 11**.

Listing 11.

```
@TestFactory
Stream<DynamicTest> dynamicSquareRootTest() {
    return Stream.of(
      new Object[][] {{2d, 4d}, {3d, 9d}, {4d, 16d}})
          .map(i -> dynamicTest("Square root test",
      () -> {
        assertEquals(i[0], Math.sqrt((double)i[1]));
      }));
}
```

The dynamicSquareRootTest method annotated with @TestFactory is not a test case, but it creates a Stream instance of the DynamicTest class, which will contain the various test implementations. For each tuple element of the stream, I execute a lambda expression that maps to an executable test case, and that test case does an assertion that tests whether the first element of the tuple equals the square root of the second element of the tuple. Isn't that elegant?

Conclusion

The JUnit team has succeeded in offering a new, redesigned version of JUnit that addresses nearly all the limitations of previous versions. Note that the JUnit 5 API is still subject to change; the team is annotating the public types with the @API annotation and assigning values such as Experimental, Maintained, and Stable.

Give JUnit 5 a spin, and keep your green bar always on!

[This article is a considerably updated version of our initial coverage of JUnit 5 in the May/June issue. —*Ed.*] </article>

Mert Çalişkan (@mertcal) is a Java Champion and coauthor of *PrimeFaces Cookbook* (Packt Publishing, first edition, 2013; second edition, 2015) and *Beginning Spring* (Wiley Publications, 2015). He is the founder of Ankara JUG, the most active Java user group in Turkey.

learn more

JUnit 5 official documentation

Another overview of this release of JUnit 5

MERT ÇALIŞKAN

Part 2: Using JUnit 5

Integrating with build tools and IDEs and running v5 tests with earlier versions

In Part 1 of this article (page 14), I wrote about the features coming in JUnit 5. In this article, I provide more details on the framework and its integration with build tools such as Maven and Gradle.

All examples in this article are based on JUnit version 5.0.0-M2, which can be found on the project home page.

Architectural Overview

Let's start with the packaging structure of JUnit 5, which was revised after its alpha release. JUnit 5 now consists of three main packages: Platform, Jupiter, and Vintage. The packaging structure and modules are illustrated in **Figure 1**, which is current as of the M2 release.

JUnit Platform is the package that provides the foundation for the Vintage and Jupiter packages. It contains the `junit-platform-engine` module, which offers a public API for integration of third-party testing frameworks such as Specsy, which is a behavior-driven development (or BDD)-style unit-level testing framework for JVM languages. The launching API that is provided for build tools and IDEs now resides under the `junit-platform-launcher` module. Prior to version 5, both IDEs and test code used the same JUnit artifact. The new, more-modular approach introduces a good separation of concerns by having build tool artifacts and the API segregated into different modules.

The `junit-platform-runner` module provides an API for running JUnit 5 tests on JUnit 4. The `junit-platform-console` module provides support for launching the JUnit platform from a console that will enable running JUnit 4 and JUnit 5 tests from the command line and printing execution results back to the console. The `junit-platform-surefire-provider` module contains the `JUnitPlatformProvider` class, which integrates with the Surefire plugin to run JUnit 5 tests via Maven. (Surefire is the Maven plugin that runs JUnit tests during the test cycle.) In addition, the `junit-platform-`

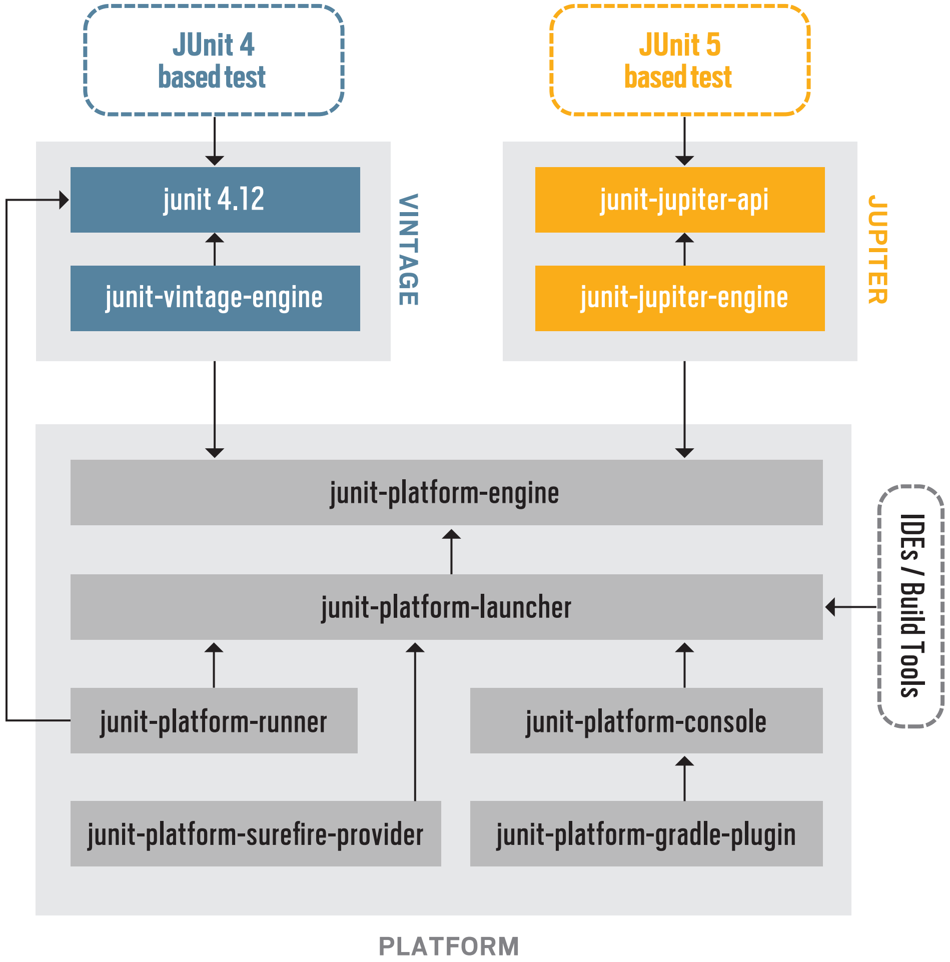


Figure 1. The JUnit 5 architecture

gradle-plugin module offers integration with Gradle builds. I describe that later in this article.

The JUnit Vintage package provides an engine for running JUnit 3 and JUnit 4 tests on JUnit 5. The junit-vintage-engine module is the engine that executes those tests. The JUnit team provided the support for a former version of the framework, and this support will encourage upgrading to JUnit 5 regardless of the version in use. In a later section, I describe the ways to run JUnit 4 tests.

JUnit Jupiter is a wrapper module of the new API and the extension model, and it also provides an engine for running JUnit 5 tests. junit-jupiter-api and junit-jupiter-engine are submodules of the project. If you have only the junit-jupiter-engine dependency defined, that suffices for executing JUnit 5 tests because the junit-jupiter-api module is a transitive dependency to the junit-jupiter-engine module.

Configuring Tools to Use JUnit 5

JUnit 5 dependency definitions are available for the Maven and Gradle frameworks. In addition, it's also possible to execute tests directly through the console. Some IDEs have already started to provide support for running JUnit 5 tests, so things look promising for the adoption of the framework.

Maven integration. The Maven dependency definition for JUnit 5 is shown in Listing 1. As I mentioned before, there is no need to define the junit-jupiter-api module, because it will be fetched as a transitive dependency when I declare junit-jupiter-engine.

Listing 1.

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.0.0-M2</version>
    <scope>test</scope>
</dependency>
```

If you want to stick with JUnit 4.x, that is also possible within JUnit 5 by defining its vintage-mode dependency, as shown in Listing 2.

Listing 2.

```
<dependency>
     <groupId>org.junit.vintage</groupId>
     <artifactId>junit-vintage-engine</artifactId>
     <version>4.12.0-M2</version>
     <scope>test</scope>
</dependency>
```

The JUnit 4.12 and junit-platform-engine transitive dependencies are retrieved automatically when vintage mode is declared. For convenience, the JUnit team aligned the version of the vintage modules with the latest available production-ready JUnit release, which was 4.12 at the time of this writing.

After defining dependencies, it's time to execute your tests by using those dependencies. Inside the Maven build cycle, maven-surefire-plugin should be defined with the junit-platform-surefire-provider dependency, as shown in Listing 3.

Listing 3.

```
<plugin>
    <artifactId>
        maven-surefire-plugin
    </artifactId>
    <version>2.19.1</version>
    <dependencies>
      <dependency>
        <groupId>
          org.junit.platform
        </groupId>
        <artifactId>
          junit-platform-surefire-provider
        </artifactId>
        <version>1.0.0-M2</version>
      </dependency>
    </dependencies>
</plugin>
```

The JUnit team developed the `junit-platform-surefire-provider` dependency, and its aim is to enable running both JUnit Vintage and JUnit Jupiter tests through Surefire's mechanism. The dependency doesn't yet support advanced parameters of Surefire, such as `forkCount` and `parallel`, but I believe the next iterations of the Surefire plugin will close that gap and will be 100 percent compatible with JUnit 5.

Gradle integration. Defining dependencies in Gradle is similar to defining them in Maven. Jupiter's engine and API definitions for Gradle are shown in Listing 4.

Listing 4.

```
dependencies {
    testCompile("org.junit.jupiter:+
                 junit-jupiter-engine:5.0.0-M2")
}
```

And the Vintage engine definition is given in Listing 5.

Listing 5.

```
dependencies {
    testCompile("org.junit.vintage:+
                 junit-vintage-engine:4.12.0-M2")
}
```

To get the JUnit Gradle plugin hooked into the build, the plugin should be declared and then applied in the configuration file, as shown in Listing 6.

Listing 6.

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath 'org.junit.platform:+
              junit-platform-gradle-plugin:1.0.0-M2'
    }
}

apply plugin: 'org.junit.platform.gradle.plugin'
```

As shown in Listing 7, it's also possible to configure the JUnit Gradle plugin with the `junitPlatform` directive, with which I can define the engines for execution (all engines are enabled by default, by the way) and test tags for inclusion or exclusion; set a naming strategy for filtering test classes for execution; specify a customized-reports directory definition; and specify a log manager configuration.

Listing 7.

```
junitPlatform {
      engines {
            // include 'junit-jupiter', 'junit-vintage'
            // exclude 'custom-engine'
      }
      tags {
            // include 'fast'
            // exclude 'slow'
      }
      // includeClassNamePattern '.*Test'
      // enableStandardTestTask true
      // below is the default reports directory
      // "build/test-results/junit-platform"
```

```
        logManager 'org.apache.logging.log4j.jul.+
                    LogManager'
}
```

Console integration. The ConsoleLauncher command-line application enables you to run the JUnit Platform directly from the console. The launcher can be executed with the Java command shown in Listing 8. Building the classpath with the needed JAR files is a prerequisite, so ensure that you have the correct version of the artifacts.

Listing 8.

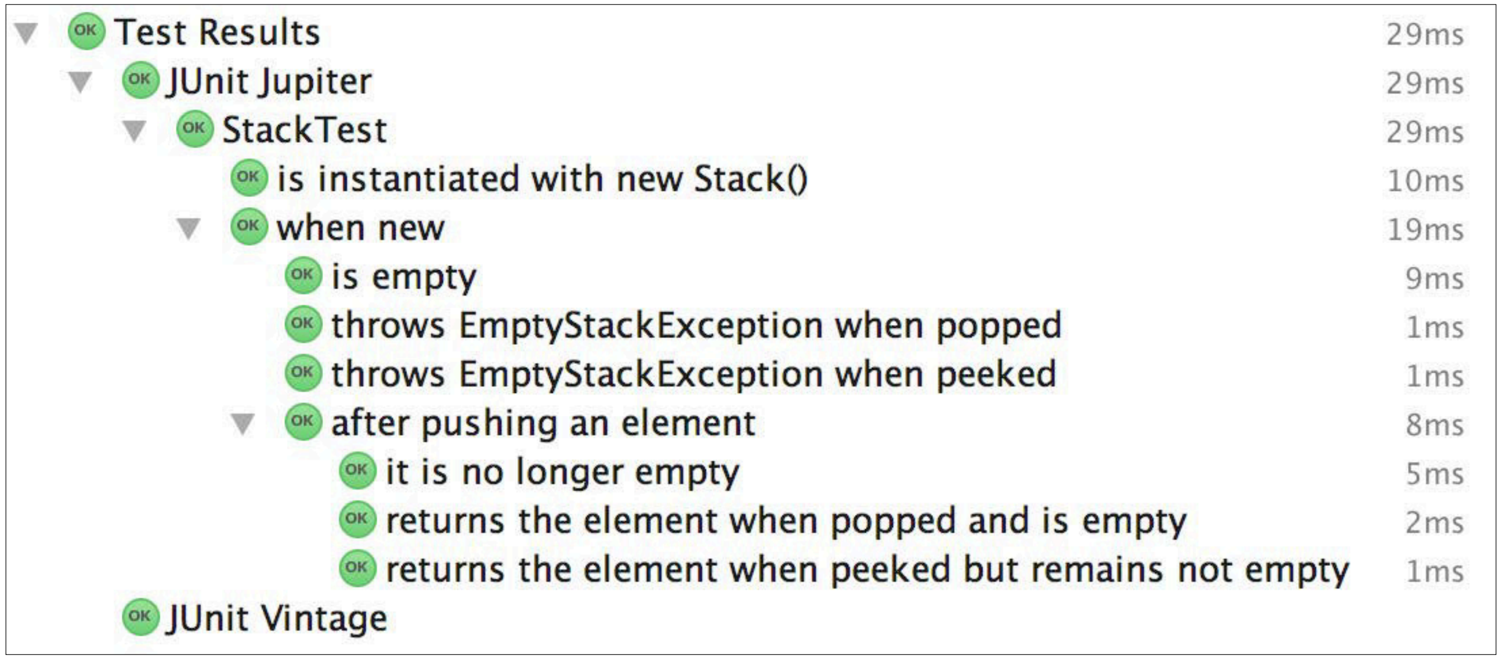
```
java -cp
  /path/to/junit-platform-console-1.0.0-M2.jar:
  /path/to/jopt-simple-5.0.2.jar:
  /path/to/junit-platform-commons-1.0.0-M2.jar:
  /path/to/junit-platform-launcher-1.0.0-M2.jar:
  /path/to/junit-platform-engine-1.0.0-M2.jar:
  /path/to/junit-jupiter-engine-5.0.0-M2.jar:
  /path/to/junit-jupiter-api-5.0.0-M2.jar:
  /path/to/opentest4j-1.0.0-M1.jar:
  org.junit.platform.console.ConsoleLauncher -a
```

[The classpath should be entered as a single line. —*Ed.*]

The -a argument specifies that all tests should be run. The -n argument can also be used to run only test classes whose fully qualified names match a regular expression. Several other options are available, although according to the documentation they are subject to change.

IDE integration. Java IDEs on the market are quickly evolving to provide robust support for running JUnit 5 tests. At the time of this writing, IntelliJ IDEA handles JUnit 5 tests with its current release and creates a tree for tests by providing support for both Jupiter and Vintage packages. Sample output for testing a series of stack operations is shown in Figure 2. The test class contains the new @Nested annotation on test classes, which enables creating hierarchies of tests, which are correctly represented in this figure.

Eclipse Neon and NetBeans 8.1 also support executing JUnit 5 tests with the help of the JUnitPlatform runner, which makes it possible to run JUnit 5 tests on the JUnit 4 platform within the IDE.



```
▼ OK Test Results                                                    29ms
   ▼ OK JUnit Jupiter                                                29ms
      ▼ OK StackTest                                                 29ms
            OK is instantiated with new Stack()                      10ms
         ▼ OK when new                                               19ms
               OK is empty                                            9ms
               OK throws EmptyStackException when popped              1ms
               OK throws EmptyStackException when peeked              1ms
            ▼ OK after pushing an element                             8ms
                  OK it is no longer empty                            5ms
                  OK returns the element when popped and is empty     2ms
                  OK returns the element when peeked but remains not empty  1ms
      OK JUnit Vintage
```

Figure 2. Output from IntelliJ for nested tests

Backward Compatibility with Vintage Mode

With the help of the JUnitPlatform class, which is an implementation of JUnit 4's Runner, it's possible to execute JUnit Jupiter-based test classes on JUnit 4. The junit-platform-runner module contains the needed JUnitPlatform class and it should be defined with the

Jupiter engine dependency, as shown in Listing 9, in Maven.

Listing 9.

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.0.0-M2</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.junit.platform</groupId>
    <artifactId>junit-platform-runner</artifactId>
    <version>1.0.0-M2</version>
    <scope>test</scope>
</dependency>
```

A sample test case implementation is given in Listing 10. As seen in the import statements, the test case is solely implemented with JUnit 5, and the defined runner makes it possible to execute the test on JUnit 4–based platforms such as Eclipse Neon.

Listing 10.

```
import org.junit.jupiter.api.Test;
import org.junit.platform.runner.JUnitPlatform;
import org.junit.runner.RunWith;

import static
    org.junit.jupiter.api.Assertions.assertTrue;

@RunWith(JUnitPlatform.class)
class SampleTest {

    @Test
    void sampleTest() {
        assertTrue(true);
    }
}
```

Conclusion

The JUnit team did a very good job with the latest release of version 5, and the new packaging structure shows that the framework has been revamped to provide a foundation for many future releases. JUnit 5 addresses nearly all the limitations of the previous version and provides better support via integration points for build tools, IDEs, and third-party testing frameworks. By leveraging the use of lambdas and with the help of new implementations such as the extension model, I believe JUnit will continue to be the most popular Java framework. </article>

Mert Çalişkan (@mertcal) is a Java Champion and coauthor of *PrimeFaces Cookbook* (Packt Publishing, first edition, 2013; second edition, 2015) and *Beginning Spring* (Wiley Publications, 2015). He is the founder of Ankara JUG, the most active Java user group in Turkey.

learn more

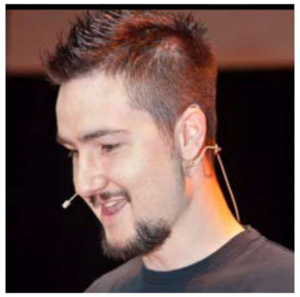
[In-depth blog on the JUnit architecture](#)
[Using JUnit 5 in IntelliJ IDEA](#)
[JUnit team on Twitter](#)

☛ “Part 1: A First Look at JUnit 5,” page 14

☛ “A Deep Dive into JUnit 5’s Extension Model,” page 25

☛ “Interview with Kent Beck,” page 36

☛ “Mutation Testing: Automate the Search for Imperfect Tests,” page 43



NICOLAI PARLOG

A Deep Dive into JUnit 5's Extension Model

The lowdown on how JUnit runs tests and how it interacts with libraries and frameworks

The next release of JUnit is version 5, indicating a major release of Java's ubiquitous testing library. The primary reasons for this major release are a new architecture that separates JUnit the tool from JUnit the platform and a new extension model that does away with key limitations of the previous architecture.

In this article, I examine the extension model, which is what third-party libraries and frameworks use to integrate with or extend JUnit. This topic will be of primary interest to writers of those tools and libraries, as well as to developers who want intimate knowledge of how JUnit works. To follow along, you will need to have a good working knowledge of JUnit 4.

I should add that application developers who spend the time to understand the extension model will be in a position to reduce boilerplate and improve the readability and maintainability of their tests.

Test Lifecycle

Extensions hook into the test lifecycle, so let's look at that first. Let's take the following test as an example:

```
// @Disabled <1.>
class LifecycleTest {

    LifecycleTest() { /* <2.> */ }

    @BeforeAll
    static void setUpOnce() { /* <4.> */ }

    @BeforeEach
    static void setUp() { /* <5.> */ }

    @Test
    // @Disabled <3.>
    void testMethod(String parameter /* <6.> */)
         { /* <7. then 8.> */ }

    @AfterEach
    static void tearDown() { /* <9.> */ }

    @AfterAll
    static void tearDownOnce() { /* <10.> */ }

}
```

Here are the steps in the lifecycle (the numbers refer to the comments above):

1. Check whether the tests in the test class (called the *container* by JUnit) should be executed.
2. An instance of the container is created.
3. Check whether the particular test should be executed. (From a lifecycle point of view, this step occurs here.

However, from a coding point of view, this is generally specified in the location marked in the preceding code.)

4. The @BeforeAll (formerly @BeforeClass) methods are called if this is the first test from the container.
5. The @BeforeEach (formerly @Before) methods are called.
6. Test method parameters are resolved. (Note that test methods can now have parameters!)
7. The test is executed.
8. Possible exceptions are handled.
9. The @AfterEach (formerly @After) methods are called.
10. The @AfterAll (formerly @AfterClass) methods are called if this is the last test from the container.

Now let's see how to interact with that lifecycle.

Extension Points

When the JUnit 5 project came together in 2015, the main designers decided on a couple of core principles, one of which was this: prefer extension points over features.

And quite literally, JUnit 5 has extension points. So when a test steps through the lifecycle described above, JUnit pauses at defined points and checks which extensions want to interact with the running test at that particular step. Here's a list of the extension points:

- ContainerExecutionCondition
- TestInstancePostProcessor
- TextExecutionCondition
- BeforeAllCallback
- BeforeEachCallback
- BeforeTestExecutionCallback
- ParameterResolver
- TestExecutionExceptionHandler
- AfterTestExecutionCallback
- AfterEachCallback
- AfterAllCallback

Note how they correspond closely to the testing lifecycle. Of these, only BeforeTestExecutionCallback and AfterTestExecutionCallback are new. They are more of a technical requirement in case an extension needs to run as closely to the test as possible, for example, to benchmark a test.

What exactly is an extension, and how does it interact with extension points? For each of the extension points, there exists a Java interface that has the same name. The interfaces are quite simple and usually have one, sometimes two, methods. At each point, JUnit gathers a lot of context information (I'll get to that shortly), accesses a list of registered extensions that implement the corresponding interface, calls the methods, and changes the test's behavior according to what is returned by the methods.

A Simple Benchmark

Before diving deeper, let's look at a simple example. Let's say I want to benchmark my tests, which, for this example, means I'll print elapsed time to the console. As you'd expect, before test execution, I store the test launch time, and after execution, I print the elapsed time.

Looking over the list of extension points, two stand out as being useful: BeforeTestExecutionCallback and AfterTestExecutionCallback. Here are their definitions:

```
public interface BeforeTestExecutionCallback
  extends Extension {

    void beforeTestExecution(
      TestExtensionContext context) throws Exception;

}

public interface AfterTestExecutionCallback
  extends Extension {

    void afterTestExecution(
      TestExtensionContext context) throws Exception;

}
```

This is what the extension looks like:

```
public class BenchmarkExtension implements
               BeforeTestExecutionCallback,
               AfterTestExecutionCallback {

    private long launchTime;

    @Override
    public void beforeTestExecution(
      TestExtensionContext context) {
       launchTime = System.currentTimeMillis();
    }

    @Override
      public void afterTestExecution(
        TestExtensionContext context) {
          long elapsedTime =
            System.currentTimeMillis() - launchTime;
            System.out.printf(
              "Test took %d ms.%n", elapsedTime);
        }

}
```

Registering Extensions

It is not sufficient to implement an extension; JUnit also has to know about it. An extension can be registered with the `@ExtendWith` annotation, which can be used on types and methods and takes the extension's class as an argument. During test execution, JUnit looks for these annotations on test classes and methods and runs all extensions it can find.

Registering an extension on a single container or method is *idempotent*—meaning registering the same extension multiple times on the same element has no additional effect. What about registering the same one on different elements?

Extensions are "inherited" in the sense that a method inherits all extensions applied to the containing type, and a type inherits all the extensions of its supertypes. They are applied *outside-in*, so, for example, a "before-each" extension that was registered with a container is executed before extensions to the same point on the executed method.

Stores are hierarchical because a store is created for each extension context, which means there is one store per node in the test tree.

The outside-in approach, as opposed to a *top-down* approach, implies that extensions adding "after" behavior are executed in reverse order. That is, extensions registered on methods are executed before those registered with the corresponding container.

Registering different extensions for the same extension point is, of course, possible as well. They are also applied outside-in in the order in which they are declared.

Registering a benchmark extension. With that knowledge, let's apply a benchmark extension:

```
// this is the way all methods are benchmarked
@ExtendWith(BenchmarkExtension.class)
class BenchmarkedTest {
    @Test
    void benchmarked() throws InterruptedException {
        Thread.sleep(100);
    }
}
```

After registering the extension on the container, JUnit applies it to all contained tests, and running benchmarked will output Test took 100 ms.

What would happen if you added another method and registered the same extension again?

```
@Test
@ExtendWith(BenchmarkExtension.class)
void benchmarkedTwice() throws InterruptedException {
    Thread.sleep(100);
    assertTrue(true);
}
```

Given the explanation presented earlier, the extension would be applied again and, indeed, you'd get the output twice.

Resolving extensions. Let's get a feeling for how registration is implemented. Whenever a test node (this could be a container or a method) is prepared for execution, it takes the AnnotatedElement that it wraps (either a class or a method) and uses reflection to access the @ExtendWith annotations before extracting the actual extension classes.

Thanks to a handy utility method and streams, this happens in a pretty inconspicuous piece of code in JUnit:

```
List<Class<? extends Extension>> extensionTypes =
    findRepeatableAnnotations(annotatedElement,
                              ExtendWith.class)
        .stream()
        .map(ExtendWith::value)
        .flatMap(Arrays::stream)
        .collect(toList());
```

That List is used to create an ExtensionRegistry, which converts the list into a set to implement idempotence. The registry not only knows the extensions on the element it is created for (let's say a method), but it also holds a reference to the registry that was created for the parent node (a container in the example). Whenever a registry is queried for extensions, it accesses its parent registry and includes the extensions applied to it in its results. The call to the parent registry likewise accesses its parent and so on.

To implement the outside-in semantics I described earlier, ExtensionRegistry offers two methods: getExtensions and getReversedExtensions. The former lists the parent's extensions before its own, thereby making it suitable for the "before" order described earlier. The latter simply inverts the result of the former, so it is called for "after" use cases.

Seamless Extensions

Applying extensions with @ExtendWith works but is pretty technical and cumbersome. Luckily, the JUnit team thought so as well, and they implemented a simple feature with powerful consequences: The utility methods that look for annotations know about *meta-annotations*, which is a term that describes annotations that are applied to other annotations.

This means that it is not necessary to annotate a type or method with @ExtendWith. It is sufficient that it bears an annotation that is either directly or, through the same process, indirectly annotated with @ExtendWith. This has important benefits for readability and enables you to write extensions that seamlessly integrate with library features. Let's look at two use cases.

Seamless benchmarks. Creating a more beautiful variant of the benchmark extension is straightforward:

```
@Target({ TYPE, METHOD, ANNOTATION_TYPE })
@Retention(RetentionPolicy.RUNTIME)
@ExtendWith(BenchmarkExtension.class)
public @interface Benchmark { }
```

Thanks to the Java ElementType specified with @Target, I can use @Benchmark on test containers and methods, as well as on other annotations, for further composition. This lets me rewrite the earlier example to look much friendlier:

```
@Benchmark
class BenchmarkedTest {

    @Test
    void benchmarked() throws InterruptedException {
        Thread.sleep(100);
    }
}
```

Notice how much simpler that is.

Compositing features and extensions. Another useful pattern enabled by JUnit meta-annotations is the composition of existing features and extensions into new, intention-revealing annotations. A simple example is `IntegrationTest`:

```
@Target(METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Benchmark
@Tag("integration")
@Test
public @interface Benchmark { }
```

This is a custom annotation that a project might create to fulfill common requirements for integration tests. In this case, all such tests are benchmarked and tagged as `integration`, which allows them to be filtered and, most importantly, annotated with `@Test`, which allows you to use `@IntegrationTest` instead of `@Test`:

```
class ServerTest {

    @IntegrationTest
    void testLogin {
        // long running test
        Thread.sleep(10000);
    }

}
```

Extension Context

One of the cornerstones of the extension model is the `ExtensionContext`, an interface with two specializations: `ContainerExtensionContext` and `TestExtensionContext`. It makes information regarding the current state of a container or test available to extensions. It also offers some APIs for interacting with the JUnit machinery. A look at its methods shows what it has to offer:

```
Optional<ExtensionContext> getParent();

String getUniqueId();
String getDisplayName();
Set<String> getTags();

Optional<AnnotatedElement> getElement();
Optional<Class<?>> getTestClass();
Optional<Method> getTestMethod();

void publishReportEntry(Map<String, String> map);

Store getStore();
Store getStore(Namespace namespace);
```

JUnit creates a tree of test nodes, and each node produces these contexts. Because the nodes have parents (for example, the node corresponding to a test class is parent to the nodes corresponding to the methods it declares), they let their extension context reference their parent's context.

To enable you to identify and filter containers and tests, these items have IDs, more-human-readable display names, and tags, which are accessed using the context methods. Very importantly, the context gives access to the class or method it was created for. This enables extensions to use reflection, which can, for example, access a test's annotations or a class's fields. Let's see this in action by slightly enriching the benchmark extension to include the test's display name in the logged message:

```
@Override
public void afterTestExecution(
  TestExtensionContext context) {
    long elapsedTime =
        System.currentTimeMillis() - launchTime;
    System.out.printf("Test '%s' took %d ms.%n",
        context.getDisplayName(), elapsedTime);
}
```

And you can go even further. Instead of crudely printing to the console, you can use JUnit's report infrastructure by calling `publishReportEntry`:

```
@Override
public void afterTestExecution(
  TestExtensionContext context) {
    long elapsedTime =
      System.currentTimeMillis() - launchTime;
    String message =
      String.format("Test '%s' took %d ms.",
        context.getDisplayName(), elapsedTime);
    context.publishReportEntry(
      createMapWithPair("Benchmark", message));
}
```

I won't discuss JUnit's reporting facility in depth, but suffice it to say that it is a way to log messages to different output sinks, such as the console or XML reports. The method `publishReportEntry` enables an extension to interact with the report. Finally, there is a data store that must be used to persist an extension's state. I'll discuss this shortly.

It is straightforward to run old JUnit 4 tests and new JUnit 5 tests side by side. **This means it is not necessary to migrate individual tests.**

As I've mentioned, JUnit is in charge of identifying and applying extensions, which implies that it is also managing instances of the extensions. How does it do that? If you are going to assign information you gathered during a run to fields, as I did in `BenchmarkExtension`, you need to know the extension's scope and lifetime.

As it turns out, that's an intentionally unspecified implementation detail. Defining a lifecycle for extension instances and tracking them during a running test suite is at best bothersome and at worst a threat to maintainability. So all bets are off! JUnit makes no guarantees whatsoever regarding the lifecycle of extension instances. Hence, they need to be stateless and should store any information on a data structure provided by JUnit for that specific purpose: the store.

The store. A store is a namespaced, hierarchical, key-value data structure. Let's look at each of these properties in turn.

To access the store via the extension context, a namespace must be provided. The context returns a store that manages entries exclusively for that namespace. This is done to prevent collisions between different extensions operating on the same node, which could lead to accidental sharing and mutation of state. (Interestingly enough, the access via namespaces can be used to intentionally access another extension's state, allowing communication and, hence, interaction between extensions, which could lead to interesting cross-library features.)

Stores are hierarchical because a store is created for each extension context, which means there is one store per node in the test tree. Each test container or test method has its own store. In much the same way as nodes “inherit” extensions, stores inherit state. To be more precise, when a node creates a store, the node hands the store a reference to its parent’s store. Thus, for example, the store belonging to a test method holds a reference to the store belonging to the test class that contains the method. Upon queries (but not edits), a store first checks its own content before delegating to its parent store. This makes a store’s state readable to all child stores.

Regarding having a key-value data structure, a store is a simplified map in which keys and values can be of any type. Here are the most essential methods:

```
interface Store {
    void put(Object key, Object value);
    <V> V get(Object key, Class<V> requiredType);
    <V> V remove(Object key, Class<V> requiredType);
}
```

The methods `get` and `remove` take a type token to prevent clients from littering their code with casts.

There is no magic there; the store simply does the casts internally. Overloaded methods without type tokens exist as well.

Stateless benchmarks. To make the benchmark extension stateless, I need a couple of things:

- A namespace for the extension to access a store
- A key for the launch time
- The ability to write to and read from the store instead of from a field

For the first two, I declare two constants:

```
private static final Namespace NAMESPACE =
    Namespace.create("org", "codefx", "Benchmark");
```

```
private static final String LAUNCH_TIME_KEY =
    "LaunchTime";
```

Reading and writing are two simple methods:

```
private static void storeNowAsLaunchTime(
  ExtensionContext context) {
      context.getStore(NAMESPACE)
          .put(LAUNCH_TIME_KEY, currentTimeMillis());
}

private static long loadLaunchTime(
  ExtensionContext context) {
      return context.getStore(NAMESPACE)
          .get(LAUNCH_TIME_KEY, long.class);
}
```

With these methods, I replace the access to the field `launchTime`, which can subsequently be removed.

The methods executed before and after each test now look as follows:

```
@Override
public void beforeTestExecution(
  TestExtensionContext context) {
    storeNowAsLaunchTime(context);
}

@Override
public void afterTestExecution(
  TestExtensionContext context) {
    long launchTime = loadLaunchTime(context);
    long runtime = currentTimeMillis() - launchTime;
    print(context.getDisplayName(), runtime);
}
```

As you can see, the new methods use the store instead of the field to persist and access the extension's state.

During the test lifecycle, JUnit pauses at each extension point, searches for all extensions that apply to the current test node, gathers context information, and calls extensions in outside-in order.

Retrofitting @Test

Let's look at a new example that leverages much of the material I've just covered.

Let's say I want to move from JUnit 4 to JUnit 5. First of all, thanks to the design of the new architecture, it is straightforward to run old and new tests side by side. This means it is not necessary to migrate individual tests, which makes what follows a little moot but no less fun.

I want to replace JUnit 4's `@Test` annotation with a new version that makes the annotated method a JUnit 5 test. I could pick JUnit 5's `@Test`, and this works in most cases: a simple search-and-replace on the import would do the trick. (Note: This is just a thought experiment, not an actual recommendation.)

But JUnit 4's optional arguments `expected` (to fail tests when a particular exception is not thrown) and `timeout` (to fail tests that run too long) are not supported in JUnit 5's annotation. JUnit 5 provides these features via assertThrows and the upcoming `assertTimeout`. But I'm looking for a way that requires no manual intervention, which precludes updating the tests to the new API.

So why not create my own `@Test` that JUnit 5 will recognize and run and that implements the desired functionality?

First things first. I'll declare a new `@Test` annotation:

```
@Target(METHOD)
@Retention(RetentionPolicy.RUNTIME)
@org.junit.jupiter.api.Test
public @interface Test { }
```

This is pretty straightforward: I just declare the annotation and meta-annotate it with JUnit 5's `@Test`, so JUnit will identify annotated methods as test methods and run them.

Expecting exceptions. To manage expected exceptions, I first need a way for the user to declare them. For this, I extend my annotation with code that is heavily inspired by JUnit 4's implementation of this feature:

```
public @interface Test {

    class None extends Throwable {
        private static final long
            serialVersionUID = 1L;
        private None() { }
    }

    Class<? extends Throwable>
        expected() default None.class;

}
```

Now, a user can use `expected` to specify which exception to expect. It defaults to `None`.

The extension itself will be found in a class called ExpectedExceptionExtension, which is shown below. To register it with JUnit, I annotate `@Test` with `@ExtendWith(ExpectedExceptionExtension.class)`.

Next, I need to actually implement the desired behavior. Here is a short description of how I can do it:

1. If a test throws an exception, check whether it is the expected exception. If it is, store the fact that the expected exception was thrown. Otherwise, store that it was the wrong one and rethrow the exception (because

the extension is not in charge of that exception).

2. After the test is executed, check whether the expected exception was thrown. If so, do nothing because everything went as planned; otherwise, fail the test.

For this logic, I need to interact with two extension points, TestExecutionExceptionHandler and AfterTestExecutionCallback, so I implement the corresponding interfaces:

```
public class ExpectedExceptionExtension
  implements TestExecutionExceptionHandler,
             AfterTestExecutionCallback {

    @Override
    public void handleTestExecutionException(
        TestExtensionContext context,
        Throwable throwable)
            throws Throwable { }

    @Override
    public void afterTestExecution(
        TestExtensionContext context) { }
}
```

Let's start with step 1 and check whether the exception is the one expected. For this, I use a small utility function, expectedException, which accesses the @Test annotation, extracts the expected exception class, and returns it in an Optional (because maybe no exception was expected). To capture the observed behavior, I create an enum, EXCEPTION, and to persist the observation in the store, I write storeExceptionStatus. With these helpers in place, I can implement the first extension point:

```
@Override
public void handleTestExecutionException(
  TestExtensionContext context, Throwable throwable)
    throws Throwable {
      boolean throwableMatchesExpectedException =
        expectedException(context)
          .filter(expected ->
                  expected.isInstance(throwable))
          .isPresent();
      if (throwableMatchesExpectedException) {
        storeExceptionStatus( context,
          EXCEPTION.WAS_THROWN_AS_EXPECTED);
      } else {
        storeExceptionStatus(context,
          EXCEPTION.WAS_THROWN_NOT_AS_EXPECTED);
        throw throwable;
      }
}
```

Note that by not rethrowing the exception, I inform JUnit that I processed it and that all is in order. Accordingly, JUnit will neither call additional exception handlers with it nor fail the test. So far so good.

Now, after the test, I need to check what happened so I can react accordingly. Another helper, loadExceptionStatus, will retrieve the state and do me a further small favor: when no exception was thrown, the extension point I implemented above will not have been called, which means no EXCEPTION instance was placed into the store. In this case, loadExceptionStatus will return EXCEPTION.WAS_NOT_THROWN. Here is the implementation:

```
@Override
public void afterTestExecution(
  TestExtensionContext context) {
    switch(loadExceptionStatus(context)) {
      case WAS_NOT_THROWN:
        expectedException(context)
```

```
                .map(expected -> new IllegalStateException(
                      "Expected exception " + expected +
                      " was not thrown."))
                .ifPresent(ex -> { throw ex; });

            case WAS_THROWN_AS_EXPECTED:
            // the exception was thrown as expected,
            // so there is nothing to do

            case WAS_THROWN_NOT_AS_EXPECTED:
            // an exception was thrown but of the
            // wrong type; it was rethrown in
            // handleTestExecutionException,
            // so there is nothing to do here
        }
    }
```

Two details of this approach are worthy of debate here.

- Is there a more appropriate exception than IllegalStateException? For example, perhaps an AssertionFailedError would be better.
- If the wrong exception was thrown, should I fail the test here?

I rethrew the exception in handleTestExecutionException, so presumably it either failed the test already or was caught by some other extension that made the test pass. So failing, it might break that other extension.

Both topics are worthwhile pursuing to finish this feature. But other than that, we're done with the extension to handle expected exceptions.

Timing out. The original timeout guaranteed that JUnit 4 would abandon a test once the specified time ran out. That requires pushing the test onto a separate thread. Unfortunately, JUnit 5 has no extension points interacting with threads, so this is not possible. One dire consequence is that there can be no extensions that run tests on specific threads, such as the event dispatch thread for Swing tests or the application thread for JavaFX tests. The JUnit team is well aware of this limitation. Let's hope they address it soon.

You could implement an alternative feature that measures how long a test ran, which implies that it must have finished, and fail it if it was above the threshold. With all I discussed so far, this should be fairly straightforward.

Conclusion

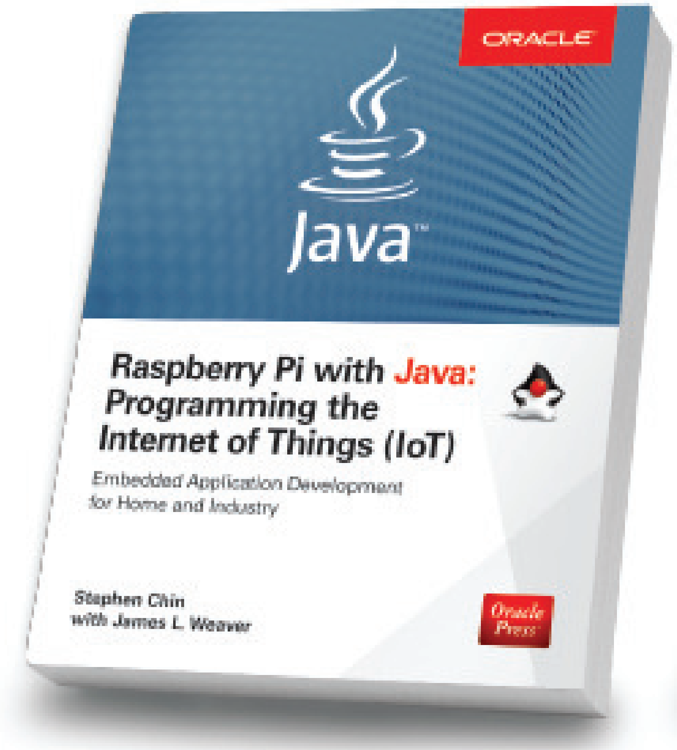
We've seen that JUnit 5 provides specific extension points, which are nothing more than small interfaces. Extension developers can implement these interfaces and register their implementations directly with @ExtendWith or, more seamlessly, with custom annotations.

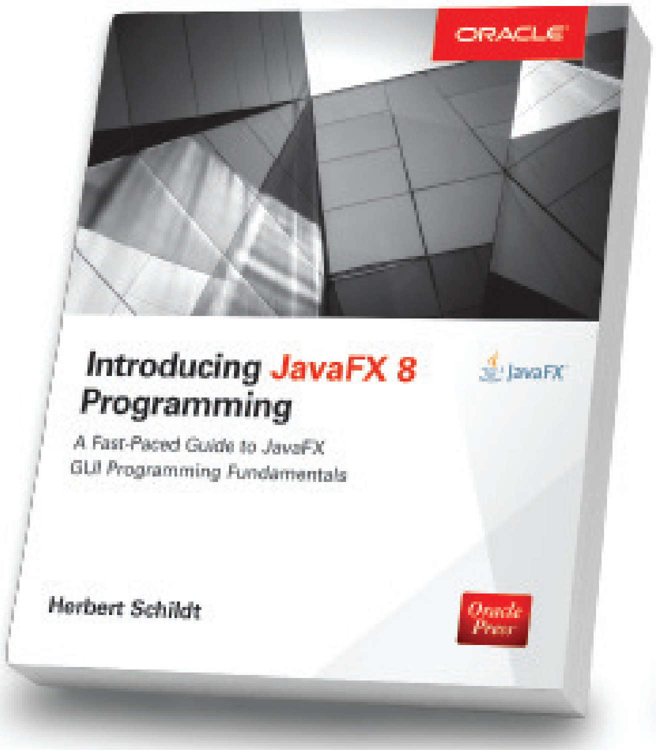
During the test lifecycle, JUnit pauses at each extension point, searches for all extensions that apply to the current test node, gathers context information, and calls extensions in outside-in order. Extensions operate on the context and whatever state they persisted in the store. JUnit reacts to the return values of the called methods and changes the test's behavior accordingly.

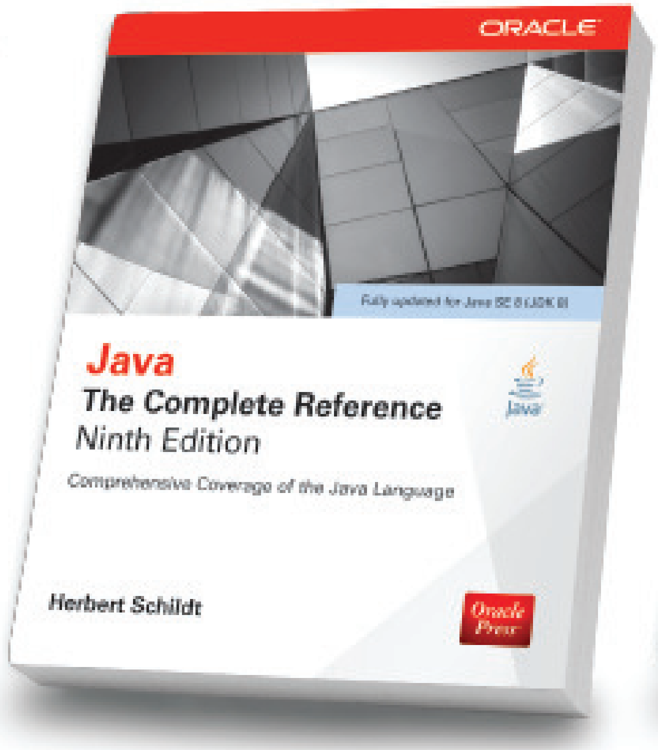
We've also seen how you can put this together for a simple benchmark extension and for a more-involved clone of JUnit 4's @Test annotation. You can find these and more examples from me on GitHub.

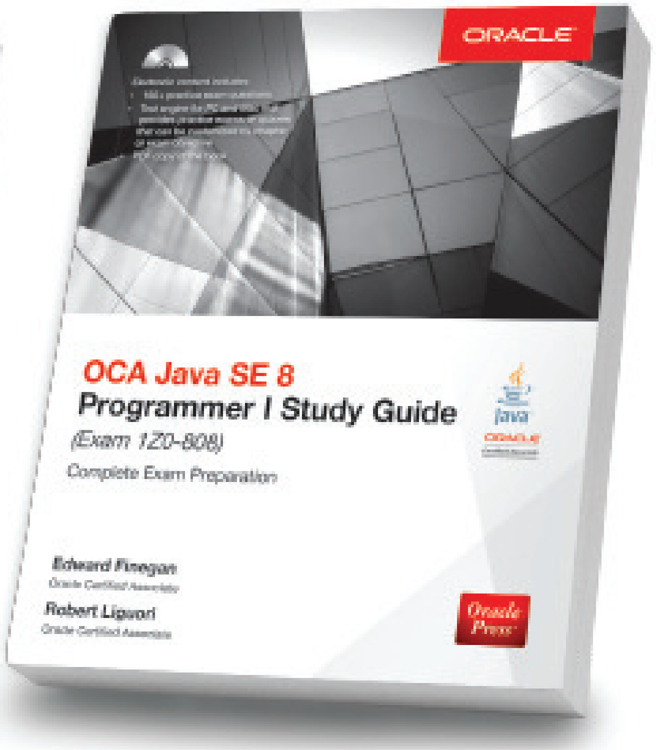
If you have any questions or remarks to share, let me know. </article>

Nicolai Parlog (@nipafx) has found his passion in software development. He codes for a living as well as for fun. Parlog is the editor of SitePoint's Java channel and blogs about software development on codefx.org. He also contributes to open source projects.


ORACLE
Java
Raspberry Pi with Java: Programming the Internet of Things (IoT)
Embedded Application Development for Home and Industry
Stephen Chin with James L. Weaver
Oracle Press


ORACLE
Introducing JavaFX 8 Programming
A Fast-Paced Guide to JavaFX GUI Programming Fundamentals
JavaFX
Herbert Schildt
Oracle Press


ORACLE
Fully updated for Java SE 8 (JDK 8)
Java
The Complete Reference
Ninth Edition
Comprehensive Coverage of the Java Language
Herbert Schildt
Oracle Press


ORACLE
OCA Java SE 8
Programmer I Study Guide
(Exam 1Z0-808)
Complete Exam Preparation
Edward Finegan
Robert Liguori
Oracle Press

Interview with Kent Beck

The parent of JUnit and creator of TDD discusses programming and testing—and how his views on testing have evolved.

ANDREW BINSTOCK

Binstock: I understand you work at Facebook these days. What is it that you do there?

Beck: I am focused on engineer education. My official title is technical coach, and that means what I do most days is pair program and talk with engineers.

Binstock: Are these typically seasoned engineers or those just recently entering the field?

Beck: All sorts. What I find if I coach a bunch of engineers at a given level, I'll start spotting patterns among whatever bottleneck they're hitting, and frankly, I get bored telling the same stories and addressing the same issues. So I'll write a course that addresses those issues. We have an organization that's very, very good at cranking lots of engineers through the course. So we have courses for new college graduates; we have a course for people making the transition to technical leadership; we have a course for technical leaders hired in from the outside, because Facebook culture is very, very different, and if you are used to leading by giving commands that other people obey, that's not going to work.

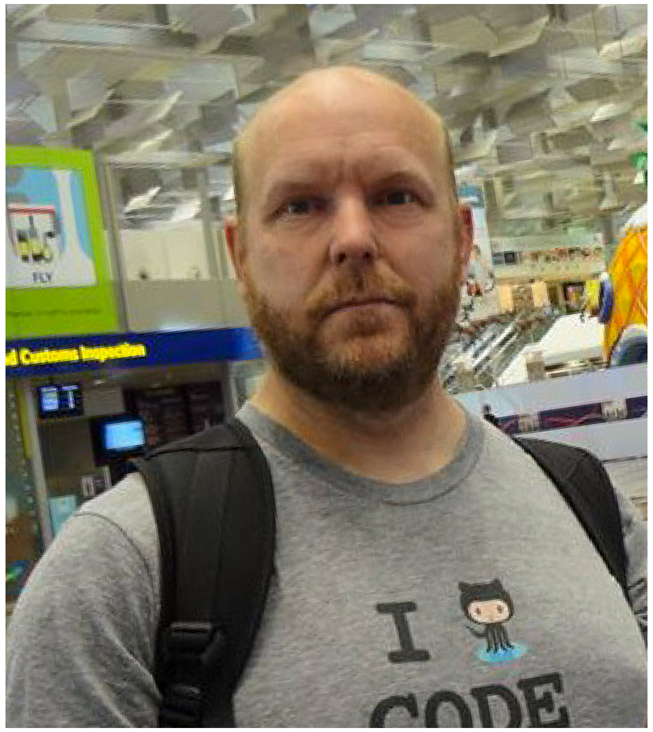
Binstock: When you're working in a place like Facebook, you're probably seeing a different kind of scaling dimension than most developers encounter. So what changes there? If I were to ask how your review of programming was informed by the concerns of scaling, what would you say is different?

Beck: It's a great question because it's really hard to boil it down, so I can give you some specifics. Logging is far more important. Performance, in some cases, is far more important. A tiny little performance regression can bring the entire site down. Because we're trying to operate very efficiently in terms of capital and also in terms of CPUs and bandwidth and everything, there's very little headroom sometimes. So, for certain teams' performance, there's a lot to lose, as well as a little bit to gain, and that's, I think, unusual. Logging is all about being able to debug after something horrible goes wrong. In classic extreme programming style, you aren't going to need it (YAGNI), so you don't write it. Well, here you are going to need it, so you do write it. Even if you don't end up ever needing it, you still need it.

Binstock: I see.

Beck: You need the option of being able to post-mortem a service, and that option's worth a lot in a way that I just had never seen before.

Binstock: How about when you commit code to the main trunk? I would imagine



Kent Beck, inventor of extreme programming and cocreator of JUnit. His work led to the popularity of developer-based testing.

PHOTOGRAPH BY BOB ADLER/
GETTY IMAGES

that the amount of testing that's applied to that code before that ever gets checked into the main build is probably significantly greater than at typical business sites. Is that true, too?
Beck: That is not true as a blanket statement. There's a principle I learned from an economics professor called reversibility. Say you have a complicated system that reacts unpredictably to stimuli. Henry Ford built these unprecedentedly large factories, complicated systems in which a tiny little change could have huge effects. So his response to that was to reduce the number of states the factory could be in by, for example, making all cars black. All cars weren't black because Henry Ford was a controlling tightwad. It was simply so that the paint shop either had paint or it didn't.

That made the whole thing easier to manage. Well, Facebook can't reduce the number of states Facebook is in. We want to keep adding more and more states. That's how we connect the world. So instead of reducing the number of states, we make decisions reversible. In Henry Ford's factory, once you cut a piece of metal, you can't uncut it. Well, we do the equivalent of that all the time at Facebook. If you make a decision reversible, then you don't need to test it with the kind of rigor that you're talking about. You need to pay attention when it rolls out and turn it off if it causes problems.
Binstock: That's an interesting alternative approach.
Beck: Well, there's a bunch of counterexamples. For example, code that handles money does go through extraordinary rigor, or the Linux kernel goes through extraordinary rigor because it's going to be deployed on hundreds of thousands of machines.

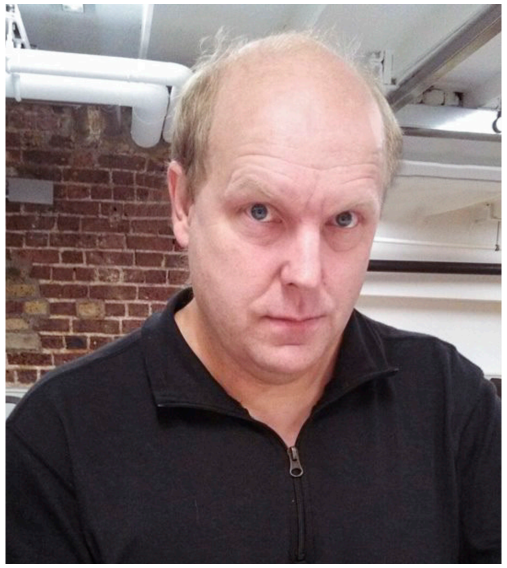
Tests are just one form of feedback, and there are some really good things about them, **but depending on the situation you're in, there can also be some very substantial costs.**

But changes to the website, you get feedback lots of different ways. You get feedback by testing it manually; you get feedback by using it internally. You get feedback by rolling it to a small percentage of the servers and then watching the metrics, and if something goes haywire, then you just turn it off.
Binstock: So nonreversible decisions get the heavy rigor and perhaps extreme testing, and everything else rides much more lightly in the saddle because of the reversibility.
Beck: Yes.

Development of JUnit

Binstock: Let's discuss the origins of JUnit. This has been documented a lot in various videos that you've made. So rather than go through it again, let me ask a few questions. How was the work initially divided between you and Erich Gamma?
Beck: We pair-programmed everything.
Binstock: So you guys were both involved throughout the entire project?
Beck: We literally did not touch the code unless we were both sitting together—for several years.
Binstock: Were you using a form of TDD at the time?
Beck: Yes, strictly. We never added a feature without a broken test case.
Binstock: OK. So how did you run the tests prior to JUnit being able to run tests?
Beck: By bootstrapping. It looked ugly at first. You might be working from the command line, and then very quickly, you get enough functionality that it becomes convenient to run the tests. Then every once in a while, you break things in a way that gives you a false positive result, and then you say, "All the tests are passing, but we're not running any tests because of whatever change we just made." Then you have to go back to bootstrapping. People should try that exer-

cise. That is an extremely informative exercise, to bootstrap a testing framework test using itself.



Binstock: The only thing you had before that was SUnit [a JUnit precursor written by Beck for Smalltalk]. That didn't seem like it was going to be very helpful in writing JUnit except on a conceptual level.
Beck: No, no, we started over from scratch.
Binstock: What role you did you have in JUnit 5? As I understand it, you were not significantly involved in this release.
Beck: Yes, I think no involvement whatsoever is probably the closest. Actually, I think at one point, they were talking about how to make two different kinds of changes in one release, and I said, by making them two different releases. So one piece of parental advice, and that was it.

Test First

Binstock: Do you still work on strictly a test-first basis?
Beck: No. Sometimes, yes.
Binstock: OK. Tell me how your thoughts have evolved on that. When I look at your book *Extreme Programming Explained*, there seems to be very little wiggle room in terms of that. Has your view changed?
Beck: Sure. So there's a variable that I didn't know existed at that time, which is really important for the trade-off about when automated testing is valuable. It is the half-life of the line of code. If you're in exploration mode and you're just trying to figure out what a program might do and most of your experiments are going to be failures and be deleted in a matter of hours or perhaps days, then most of the benefits of TDD don't kick in, and it slows down the experimentation—a latency between "I wonder" and "I see." You want that time to be as short as possible. If tests help you make that time shorter, fine, but often, they make the latency longer, and if the latency matters and the half-life of the line of code is short, then you shouldn't write tests.
Binstock: Indeed, when exploring, if I run into errors, I may backtrack and write some tests just to get the code going where I think it's supposed to go.
Beck: I learned there are lots of forms of feedback. Tests are just one form of feedback, and there are some really good things about them, but depending on the situation you're in, there can also be some very substantial costs. Then you have to decide, is this one of these cases where the trade-off tips one way or the other? People want the rule, the one absolute rule, but that's just sloppy thinking as far as I'm concerned.
Binstock: Yes, I think perhaps one of the great benefits of more than two decades of programming experience is the great distrust in one overarching rule that's unflinchingly and unbendingly applied.
Beck: Yes. The only rule is think. IBM had that right.
Binstock: I recall you saying that certain things, like getters and setters, really don't have to be written test first.
Beck: I think that's more specific than what I said. It's always been my policy that nothing has to be tested. Lots of people write lots of code without any tests, and they make a bunch of money, and they serve the world. So clearly, nothing has to be tested.

There are lots of forms of feedback, and one of the factors that goes into the trade-off equation for tests is: what's the likelihood of a mistake? So if you have a getter, and it's just a getter, it never changes. If you can mess that up, we have to have a different conversation. Tests are not going to fix that problem.
Binstock: When you originally formulated the rules for TDD, one of the cornerstones was that each iteration should have

the smallest possible increment of functionality. Where did that view come from? What was important about the smallest possible increment?

Beck: If you have a big, beautiful Italian salami, and you want to know how long it's going to take to eat the whole thing, an effective strategy is to cut off a slice and eat it and then do the arithmetic. So, 1 millimeter takes me 10 seconds, then 300 millimeters are going to take me 3,000 seconds—now maybe more, maybe less. There may be positive feedback loops, negative feedback loops, other things to change that amount of time, but at least you have some experience with it.

Literate programs just don't maintain very well because there's so much coupling between the prose and diagrams and the code.

The difference between a journeyman programmer and a master, from my perspective, is that the master never tries to eat the whole salami at once. Number one, they always take some big thing, and they put it into slices. That's the first skill—figuring out where you can slice it.

The second skill is being creative about the order in which you consume the slices, because you might think you have to go left to right, but you don't. Somebody says, "Well, you have to write the input code before you can write the output code." I say, "I respectfully disagree. I can build a data structure in memory and write the output code from that." So I can do input and then output, or output and then input.

If I have *n* slices, I have *n*-factorable permutations of those slices, some of which don't make any sense but many of which do. So the two skills of the master programmer are slicing thinner slices and considering more permutations of the slices as the order of implementation. Neither of those skills ever reaches any kind of asymptote. You can always make thinner slices, and you can always think of more orders in which to implement things to serve different purposes.

If you tell me I have a demo on Friday, I implement things in a different order than if you tell me I have to run a load test on Friday, same project. I'm going to slice it differently, and I'm going to implement the slices in a very different order depending on what my next goal is.

Binstock: So the smallest possible increment is a general rule to apply when there are other factors that don't suggest thicker slices?

Beck: I don't believe in the smallest possible slice. I believe in figuring out how to make the slices smaller. As small as I think I've gotten the slices, I always find some place, some way to make them half that size, and then I kick myself: "Why didn't I think of this before? That's not one test case; that's three test cases." Then progress goes much more smoothly.

Binstock: Well, if it hews very closely to the single responsibility principle, it seems to me that you could have the same dynamic there, where methods do just very, very small operations, and then you have to string together thousands of little tiny BB-sized methods, and figure out how to put them together.

Beck: That would be bad because if you had to open 40 or 50 classes, I would argue that violated cohesion at some point, and that's not the right way to factor it out.

Binstock: I think we're heading the same way—that there is a limit at which cutting the slice even thinner doesn't add value and starts to erode other things.

Beck: Well, today, and then I sleep on it, and I wake up in the morning and go, "Why didn't I think of that? If I slice it sideways instead of up and down, clearly it works better." Something about walking out the door on Friday afternoon and getting in my car to go home—that was the trigger for me.

The Coding Process

Binstock: Some years ago, I heard you recommend that when developers are coding, they should keep a pad of paper and a pen beside them and write down every decision that they make about the code as they're doing it.

You suggested that we would all be startled by the number of entries we would make, and that is exactly what happened with me. The more I looked at those lists that I put together, the more I realized that when interruptions occur, the ability to reconstruct the entire world view that I had before the interruption occurred depends a lot on being able to remember all of these microdecisions that have been made. The longer the gap, the more difficult it is even to consult the list to get those microdecisions back.

I'm wondering, have your thoughts on recording those microdecisions evolved in any way in which you can make that list useful rather than just having it be an exercise in coding awareness?

Beck: No, no, it hasn't. One of the things—and I've written about this—is that I'm having memory problems. So I have trouble holding big complicated, or even small, programs in my head. I can be a pair-programming partner just fine because I can rely on my partner's memory, but me sitting down and trying to write a big complicated program is just not something I can do anymore.

I can still program, though, on the UNIX command line because I can see the whole thing. So as long as it's a one-liner and I can build it, like, one command at a time, then I can accomplish programming tasks, but it's not maintainable code. It's all one-off codes. I do a lot of data mining. So if you said, build a service to do X, that's just not—different people age in different ways at different rates, and so on—but that's just not something that I can independently take off and do anymore, which is frustrating as hell.

Binstock: As I think about what you're talking about and I think about my own efforts to record those microdecisions, there's a certain part of me that has a new appreciation for things like Knuth's Literate Programming where you can actually, in the comments, capture what it is you're doing, what you're trying to do, and what decisions you've made about it. Actually, I worked that way for a while after hearing your discussion of this particular discipline. In some ways, it was helpful. In other ways, it created a lot of clutter that ultimately I had to go back and pull out of the comments. So the only reason I brought it up was just to see if you had gone any further with that.

> **I'm a big believer in getting rid of textual source code** and operating directly on the abstract syntax trees.

Beck: What I find with literate programs is they just don't maintain very well because there's so much coupling between the prose and diagrams and the code. I've never put it in those terms before, but that's exactly right. If I make a change to the code, not only do I have to change the code and maybe the test, but I also have to change these four paragraphs and those two diagrams, and I have to regather this data and render it into a graph again. So it's not efficiently maintainable. If you had code that was very stable and you wanted to explain it, then that wouldn't come into play, and it would make sense again.

Binstock: I had a conversation with Ward Cunningham in which he talked about pair programming with you many years ago and how surprised he was by how frequently you guys would come to a decision point and the tool by which you moved forward was by asking, "What is the simplest possible thing we can do that will get us past this particular point?" If you always work from the simplest possible thing, do you not, at some point, have to go back and refactor things so that you have code that you can be proud of rather than code that short-circuits the problem? How do you balance those two things?

Beck: Sure. So I don't get paid to be proud. Like in JUnit, we wrote code that we'd be proud of or we didn't write the code. We could make that trade-off because we had no deadlines and no paying customers.

But on a regular basis, if I'm not proud of the code but my employer is happy with the results, yes. They call it work, and I get paid to do it. So there are other reasons to clean up.

The answer is sure, you're going to make locally optimized decisions because you just don't know stuff, and then you do know stuff and once you learn, then you're going to realize the design should've been like this and this and this instead. Then you have to decide when, how, and whether to retrofit that insight into your current code. Sometimes you do and sometimes you don't.

But I don't know what the alternative is. People say, "Well, aren't you going to have to go refactor?" Well, sure. So what's the alternative?

I remember I gave a workshop in Denmark, and I gave a day-long impassioned speech about the beauties of iteration. At the end of the day, this guy had been sitting in the front row the entire day looking at me with an increasingly troubled expression—worse, and worse, and worse. He finally raised his hand just before the time was up, and he said, "Wouldn't it be easier to do it right the first time?" I wanted to hug him. I said, "With all the compassion I have in me, yes, it would. I don't have any response other than that."

Binstock: Lovely question!

Beck: I sat next to Niklaus Wirth on an airplane once. I talked to the agent. I told him we were colleagues and would he please move me, and so I'm like a stalker—I fanboy'ed him. I don't mind. If you get a chance to sit next to Niklaus Wirth, you're going to do it. So we got to talking, and I told him about TDD and incremental design, and his response was, "I suppose that's all very well if you don't know how to design software."

Binstock: That sounds like the type of thing Wirth was known to say.

Beck: You have to say, "Well, yes, I don't know how to—congratulations! You do know. I don't. So what am I supposed to do? I can't pretend I'm you."

Testing Today

Binstock: Let's discuss microservices. It seems to me that test-first on microservices would become complicated in the sense that some services, in order to function, will need the presence of a whole bunch of other services. Do you agree?

Beck: It seems like the same set of trade-offs about having one big class or lots of little classes.

Binstock: Right, except I guess, here you have to use an awful lot of mocks in order to be able to set up a system by which you can test a given service.

Beck: I disagree. If it is in an imperative style, you do have to use a lot of mocks. In a functional style where external dependencies are collected together high up in the call chain, then I don't think that's necessary. I think you can get a lot of coverage out of unit tests.

Binstock: Today, the UI is so much more important than at any previous time. How did you unit-test UIs in the past and today? Were you using things like FitNesse and other frameworks, or were you just eyeballing the results of the tests?

Beck: I never had a satisfactory answer. Let me put it that way. I tried a bunch of stuff. I built integration testing frameworks, I used other people's tools, I tried different ways of summarizing what a UI looked like in some test-stable way, and nothing worked.

Binstock: Today, you're pretty much in the same position, aren't you?

Beck: Yes, I haven't seen anything that fundamentally changes. It's all about false positives and false negatives. What's the rate at which your tests say everything's OK, and everything's broken. That damages your trust in your tests. How often does the testing framework say something's wrong, and everything's fine? Very often, one pixel changes

color very slightly, and then the tests all break, and you have to go through one at a time and go, “Oh, yeah, no, this is fine.” And you’re not going to do that very many times before you just blow off the tests.
Binstock: The cost is the lost time and the lost trust.
Beck: Yes.

If I’m a little activated [when coding], then I’ll listen to something soothing. **And my go-to for that is Thomas Tallis’ *The Lamentations of Jeremiah.***

Coding Environment

Binstock: What does your preferred programming environment look like today, whether that’s home or work?
Beck: The things I do that look like programming, I do entirely on either UNIX command line or in Excel.
Binstock: In Excel?
Beck: Yes, because I can see everything.
Binstock: How do you mean?
Beck: So, like the transformations, I do the data transformations, like numbers to numbers on the UNIX command line, and then I render them into pictures using Excel.
Binstock: When you’re programming things that are not related to data mining, you mentioned earlier that you still use Smalltalk for exploratory things.
Beck: Yes, the advantage of Smalltalk for me is I memorized the API long enough ago that I still have access to all those details.
Binstock: Do you typically work with multiple screens?
Beck: Yes, the more pixels, the better. It was a great Terry Pratchett quote, he says, “People ask me why I have six screens attached to my Mac, and I tell them it’s because I can’t attach eight screens.” Oculus or some kind of virtual reality is just going to blow that out of the water, but nobody knows how.
Binstock: We’ll have to go through a number of iterations of things like that before virtual reality actually finds a role that’ll help with the coding.
Beck: Yes, I’m a big believer in getting rid of textual source code and operating directly on the abstract syntax trees. I did an experimental code editor called Prune with my friend Thiago Hirai. It looked like a text editor and it rendered as a text editor would render, but you could only do operations on the abstract syntax trees, and it was much more efficient, much less error-prone. It required far less cognitive effort. That convinced me that’s the wave of the future, and I don’t know if it’s going to be in 5 years or 25 years, but we’re all going to be operating on syntax trees sometime soon.
Binstock: Yes, of all the things that have changed and moved forward, the requirement that we still code at an ink-and-paper level hasn’t really moved forward very much.
Beck: No, we’re coding on punch cards. It’s rendered one on top of the other, but it’s the same darn stuff.
Binstock: The initial place of programmer activity hasn’t evolved very much at all. Despite having wonderful IDEs and things of that sort, the physical act is still very much the same. One last thing, I know you’re a musician. Do you listen to music when you code?
Beck: Yes.
Binstock: What kind of music do you find that you enjoy most coding to?
Beck: I use it to kind of regulate my energy level, so if I’m a little activated, then I’ll listen to something soothing. And my go-to for that is Thomas Tallis’ *The Lamentations of Jeremiah*, which is a very flowing vocal quartet kind of medieval music. If I’m a little low and I need picking up, then I listen to go-go music, which is an offshoot of funk native to Washington DC.
Binstock: OK. I’ve never heard of that.
Beck: That’s my upping music.
Binstock: Wonderful! Thank you! </article>

HENRY COLES

Mutation Testing: Automate the Search for Imperfect Tests

Locate incorrect and incomplete unit tests with pitest.

If you've written any code in the last week, you've most likely also written a unit test to go with it. You're not alone. These days, it's rare to find a codebase without unit tests. Many developers invest a lot of time in their tests, but are they doing a good job?

This question began to trouble me seven years ago while I was working on a large legacy codebase for a financial services company. The code was very difficult to work with, but was a core part of the business and needed to be constantly updated to meet new requirements.

A lot of my team's time was spent trying to wrestle the code into maintainable shape. This made the business owners nervous. They understood that the team had a problem and needed to make changes, but if the team introduced a bug it could be very costly. The business owners wanted reassurance that everything was going to be all right.

The codebase had a lot of tests. Unfortunately, the team didn't need to examine them very closely to see that the tests were of no better quality than the code they tested. So, before the team members changed any code, they first invested a lot of effort in improving the existing tests and creating new ones.

Because my team members always had good tests before they made a change, I told the business owners not to worry: if a bug were introduced while refactoring, the tests would catch it. The owners' money was safe.

But what if I were wrong? What if the team couldn't trust the test suite? What if the safety net was full of holes? There was also another related problem.

As the team members changed the code, they also needed to change the tests. Sometimes the team refactored tests to make them cleaner. Sometimes tests had to be updated in other ways as functionality was moved around the codebase. So even if the tests were good at the point at which they were written, how could the team be sure no defects were introduced into tests that were changed?

For the production code, the team had tests to catch mistakes, but how were mistakes in the test code caught? Should the team write tests for the tests? If that were done, wouldn't the team eventually need to write tests for the tests for the tests—and then tests that tested the tests that tested the tests that tested the tests? It didn't sound like that would end well, if it ever ended at all.

Fortunately, there is an answer to these questions. Like many other teams, my team was using a code coverage tool to measure the branch coverage of the tests.

The code coverage tool would tell which bits of the codebase were well tested. If tests were changed, the team just had to make sure there was still as much code coverage as before. Problem solved. Or was it?

There was one small problem with relying on code coverage in this way. It didn't actually tell the team anything about whether the code was tested, as I explain next.

What's Wrong with Code Coverage?

The problem is illustrated by some of the legacy tests I found within the code. Take a contrived class such as this:

```
class AClass {
  private int count;

  public void count(int i) {
    if (i >= 10) {
      count++;
    }
  }

  public void reset() {
    count = 0;
  }

  public int currentCount() {
    return count;
  }
}
```

I might find a test that looked like this:

```
@Test
public void testStuff() {
  AClass c = new AClass();
  c.count(11);
  c.count(9);
}
```

This test gives 100 percent line and branch coverage, but tests nothing, because it contains no assertions. The test executes the code, but doesn't meaningfully test it. The programmer who wrote this test either forgot to add assertions or wrote the test for the sole purpose of making a code coverage statistic go up. Fortunately, tests such as this are easy to find using static analysis tools.

I also found tests like this:

```
@Test
public void testStuff() {
  AClass c = new AClass();
  c.count(11);
  assert(c.currentCount() == 1);
}
```

The programmer has used the assert keyword instead of a JUnit assertion. Unless the test is run with the -ea flag set on the command line, the test can never fail. Again, bad tests such as this can be found with simple static analysis rules.

Unfortunately, these weren't the tests that caused my team problems. The more troubling cases looked like this:

```
@Test
public void shouldStartWithEmptyCount() {
    assertEquals(0,testee.currentCount());
}

@Test
public void shouldCountIntegersAboveTen() {
    testee.count(11);
    assertEquals(1,testee.currentCount());
}

@Test
public void shouldNotCountIntegersBelowTen() {
    testee.count(9);
    assertEquals(0,testee.currentCount());
}
```

You can't rely on code coverage tools to tell you that code has been tested.

These tests exercise both branches of the code in the count method and assert the values returned. At first glance, it looks like these are fairly solid tests. But the problem wasn't the tests the team had. It was the tests that the team didn't have.

There should be a test that checks what happens when exactly 10 is passed in:

```
@Test
public void shouldNotCountIntegersOfExcelty10() {
    testee.count(10);
    assertEquals(0,testee.currentCount());
}
```

If this test doesn't exist, a bug could accidentally be introduced, such as the one below:

```
public void count(int i) {
    if (i > 10) { // oops, missing the =
        count++;
    }
}
```

A small bug like this could have cost the business tens of thousands of dollars every day it was in production until the moment that it was noticed and fixed.

This kind of problem can't be found by static analysis. It might be found by peer review, but then again it might not. In theory, the code would never be written without a test if test-driven development (TDD) were used, but TDD doesn't magically stop people from making mistakes.

So you can't rely on code coverage tools to tell you that code has been tested. They're still useful, but for a slightly different purpose. They tell you which bits of code are definitely not tested. You can use them to quickly see which code definitely has no safety net, in case you wish to change it.

Better Coverage with Mutation Testing

One of the things I used to do after writing a test was double-check my work by commenting out some of the code I'd just implemented, or else I'd introduce a small change such as changing <= to <, as in the earlier example. If I ran my test and it didn't fail, that meant I'd made a mistake.

This gave me an idea. What if I had a tool that made these changes automatically? That is, what if there were a tool that added bugs to my code and then ran the tests? I would know that any line of code for which the tool created a bug but the test didn't fail was not properly tested. I'd know for certain whether my test suite was doing a good job.

Like most good ideas, it turned out that I wasn't the first to have it. The idea had a name—*mutation testing*—and it was first invented back in the 1970s. The topic had been researched extensively for 40 years, and the research community had developed a terminology around it.

The different types of changes I made by hand are called *mutation operators*. Each operator is a small, specific type of change, such as changing >= to >, changing a 1 to a 0, or commenting out a method call.

When a mutation operator is applied to some code, a *mutant* is created. When tests are run against a mutant version of the code, if one of the tests fails, the mutant was "killed." If no tests fail, the mutant survived.

The academics examined the various types of mutation operators that were possible, looked at which were more effective, and explored how well test suites that detected these artificial bugs could detect real bugs. They also produced several automated mutation testing tools, including some for Java.

So why haven't you heard about mutation testing before? Why aren't all developers using mutation testing tools? I'll talk about one problem now and the other a little later on.

The first problem is straightforward: mutation testing is computationally very expensive. In fact, it's so expensive that until 2009, most academic research looked only at toy projects with less than a hundred lines of code. To understand why it's so expensive, let's look at what a mutation testing tool needs to do.

Imagine you are performing mutation tests on the Joda-Time library, which is a small library for handling dates and times. This library has about 68,000 lines of code and about 70,000 lines of test code. It takes about 10 seconds to compile the code and about 16 seconds to run the unit tests.

It's not possible to perform a mutation test when you have a failing test, because doing that would mistakenly appear to kill any mutants that it covered.

Now, imagine that your mutation test tool introduces a bug on every seventh line of code. So you'd have about 10,000 bugs. Each time you change a class to introduce a bug, you need to compile the code. Perhaps that would take one second. So that would be 10,000 seconds of compilation time to produce the mutations (this is called the *generation cost*), which is more than two and a half hours. You also need to run the test suite for each mutant. That's 160,000 seconds, which is more than 44 hours. So performing mutation testing on the Joda-Time library would take almost two days.

Many of the early mutation testing tools worked exactly like this hypothetical tool. You still occasionally find people trying to sell tools that work like this, but using such a tool is clearly not practical.

When I became interested in mutation testing, I looked at the available open source tools. The best one I could find was Jumble. It's faster than the simplistic tool I described above, but it was still quite slow. And it had other problems that made it difficult to use.

I wondered if I could do better. I already had some code that seemed like it might help—it was code for running tests in parallel. It ran tests in different class loaders so that when state that was stored in static variables in legacy code was changed, the running tests wouldn't interfere with each other. I called it Parallel Isolated Test (or PIT).

After many evenings of experimentation, I managed to do better. My PIT mutation testing tool could analyze 10,000 mutations in the Joda-Time library in about three minutes.

Introducing Pitest

My tool has kept the initials of the codebase from which it grew, but it's now also known as "pitest," and it is used all over the world.

It's used for academic research and for some exciting safety-critical projects, such as for testing the control systems of the Large Hadron Collider at CERN. But mainly it is used to help test the kind of nonsafety-critical code that most developers produce every day. So how does it manage to be so much faster than the earlier system?

First, it copied a trick from Jumble. Instead of spending two and a half hours compiling source code, pitest modifies bytecode directly. This allows it to generate hundreds of thousands of mutants in subsecond time.

But, more importantly, it doesn't run all the tests against each mutant. Instead, it runs only the tests that might kill a mutant. To know which tests those might be, it uses coverage data.

The first thing pitest does is collect line-coverage data for each test so that it knows which tests execute which lines of code. The only tests that could possibly kill a mutant are the ones that exercise the line of code the mutation is on. Running any other tests is a waste of time.

Pitest then uses heuristics to select which of the covering tests to run first. If a mutant can be killed by a test, pitest usually finds the killing test in one or two attempts.

The biggest speedup is achieved when you have a mutant that is not exercised by any test. With the traditional approach, you'd have to run the entire test suite to determine that the mutant could not be killed. With the coverage-based approach, you can determine this instantly with almost no computational cost.

The most effective time to perform mutation tests on your code is when you write the code.

Line coverage identifies code that is definitely not tested. If there is no test coverage for the line of code where a mutant is created, then none of the tests in the suite can possibly kill it. Pitest can mark the mutant as surviving without doing any further work.

Using Pitest

Setting up pitest for your project is straightforward. IDE plug-ins have been built for Eclipse and IntelliJ IDEA, but personally I prefer to add mutations from the command line using the build script. Some very useful features of pitest are accessible only in this way, as you'll see in a moment.

I normally use Maven as my build tool, but pitest plugins also exist for Gradle and Ant.

Setting up pitest for Maven is straightforward. I usually bind pitest to the test phase using a profile named `pitest`. Then pitest can be run by activating the profile with `-P`, as shown here:

```
mvn -Ppitest test
```

As an example, I've created a fork of the Google assertion library Truth on GitHub, and I added pitest to the build. You can see the relevant section of the project object model (POM) file here.

Let's go through it step by step.

`<threads>2</threads>` tells pitest to use two threads when performing mutation testing. Mutation testing usually scales well, so if you have more than two cores, it is worth increasing the number of threads.

`<timestampedReports>false</timestampedReports>` tells pitest to generate its reports in a fixed location.

`<mutators><value>STRONGER</value></mutators>` tells pitest to use a larger set of mutation operators than the default. This section is commented out in the POM file at the moment. I'll enable it a little later on. If you're just starting out with mutation testing on your own project, I suggest you also stick with the defaults at first.

The pitest Maven plugin assumes that your project follows the common convention of having a group ID that matches your package structure; that is, if your code lives in packages named `com.mycompany.myproject`, it expects the group ID to be `com.mycompany.myproject`. If this is not the case, you might get an error message such as the following when you run pitest:

```
No mutations found. This probably means there is an issue
with either the supplied classpath or filters.
```

Google Truth's group name doesn't match the package structure, so I added this section:

```
<targetClasses>
  <param>com.google.common.truth*</param>
</targetClasses>
```

Note the * at the end of the package name.

Pitest works at the bytecode level and is configured by supplying globs that are matched against the names of the loaded classes, not by specifying the paths to source files. This is a common point of confusion for people using it for the first time.

Another common problem when setting up pitest for the first time is this message: `All tests did not pass without mutation when calculating line coverage. Mutation testing requires a green suite.`

This message can occur if you have a failing test. It's not possible to perform a mutation test when you have a failing test, because doing that would mistakenly appear to kill any mutants that it covered. Sometimes you'll also get the message when all the tests pass when run normally with `mvn test`. If this happens, there are a few possible causes.

Pitest tries to parse the configuration of the Surefire test runner plugin and convert the configuration to options that pitest understands. (Surefire is the plugin that Maven uses by default to run unit tests. Often no configuration is required, but sometimes tests need some special configuration in order to work, which must be supplied in the pom.xml file.)

Unfortunately, pitest can't yet convert all the possible types of Surefire configuration. If your tests rely on system properties or command-line arguments being set, you need to specify them again in the pitest configuration.

Another problem that's more difficult to spot is order dependencies in the tests. Pitest runs your tests many times in many different sequences, but you might have a test that fails if certain other tests run before it.

For example, if you have a test called `FooTest` that sets a static variable in a class to false, and you have another test called `BarTest` that assumes that the variable is set to true, `BarTest` will pass if it is run before `FooTest` but fail if it is run afterward. By default, Surefire runs tests in a random but fixed order. The order changes when a new test is added, but you might never have run the tests in an order that reveals the dependency. When pitest runs the tests, the order it uses might reveal the order dependency for the first time.

Test-order dependencies are very hard to spot. To avoid them, you can make tests defensively set shared state on which they depend to the right value when they start, and make them clean up after themselves when they finish. But by far the best approach is to avoid having shared mutable state in your program in the first place.

Finally, the setup for using the Google Truth library includes this section:

```
<excludedClasses>
  <param>
    *AutoValue_Expect_ExpectationFailure
  </param>
</excludedClasses>
```

This configuration prevents all classes whose name ends in `AutoValue_Expect_ExpectationFailure` from having mutations seeded into them. These classes are autogenerated by the Google Truth build script. There is no value in performing mutation testing on them, and any mutations that are created would be difficult to understand because you do not have the source code.

Pitest also provides other ways to exclude code from being mutation-tested. Details can be found on the pitest website.

Understanding the Pitest Report

Let's do a sample run and look at the result it generates. To begin, check out the source code for the Google Truth library, and run pitest using Maven:

```
mvn -Ppitest test
```

It should take about 60 seconds once Maven has finished downloading the dependencies. After the run, you'll find an HTML report in the `target/pitReports` directory. For the Truth project, you'll find the report under `core/target/pitReports`.

The pitest report looks very similar to the reports that standard coverage tools produce, but it contains some extra information. Each package is listed with its overall line coverage and its mutation score shown side by side.

You can drill down into each source file to get a report such as the one shown in **Figure 1**.

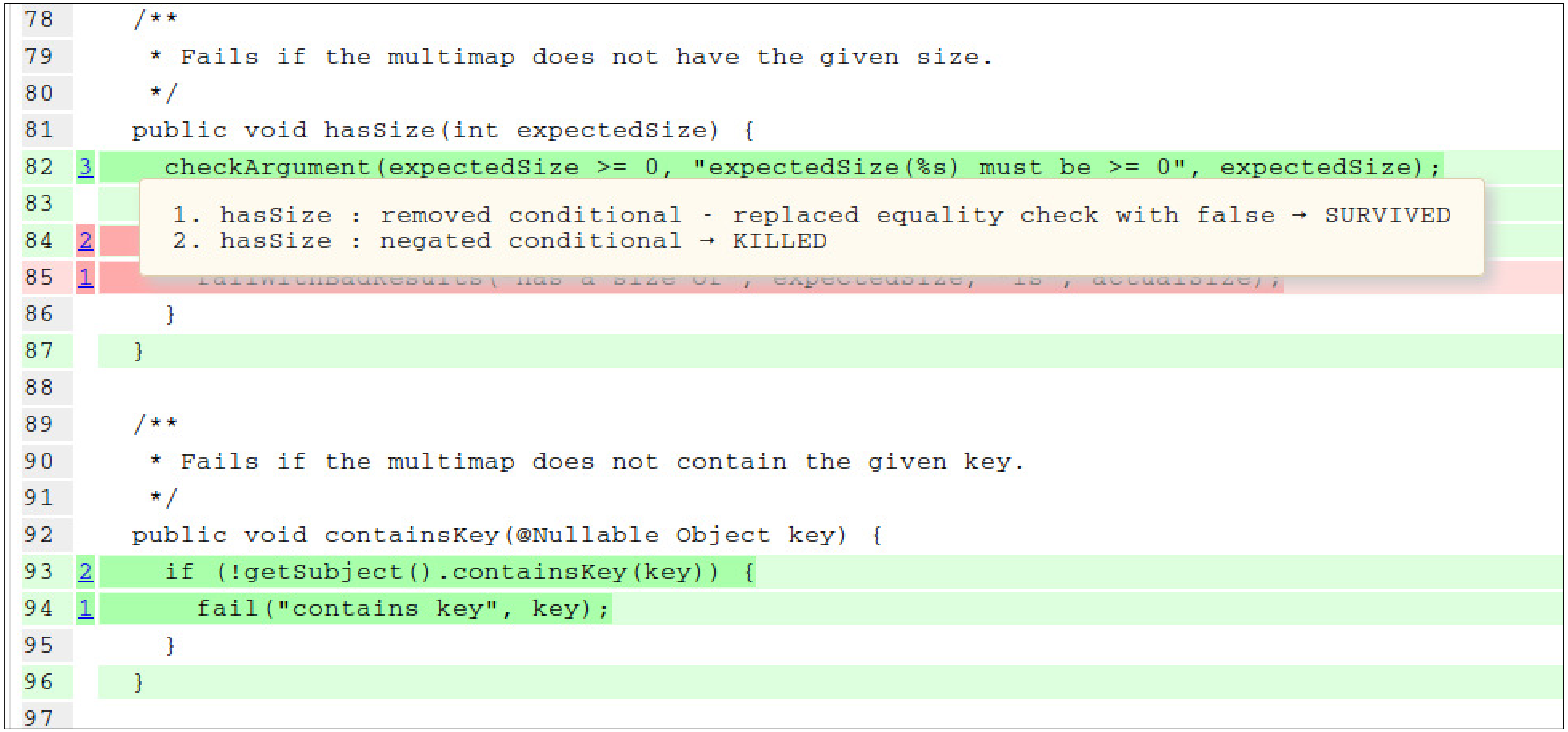
Line coverage is shown as blocks of color that span the width of the page. Green indicates that a line is executed by tests, and red indicates that it is not.

The number of mutations created on each line is shown between the line number and the code. If you hover over the number, you'll get a description of the mutations that were created and their status. If all the mutants were killed, the code is shown in darker green. If one or more of them survived, the code will be highlighted in red.

There's additional useful information at the bottom of the report: a list of all the tests that were used to challenge the mutants in this file and how long each of them took to run. Above this is a list of all the mutations. If you hover over them, you'll see the name of the test that killed the mutant.

Google Truth was developed without using pitest or any other mutation testing tool and, on the whole, the team that developed it did a very good job. A mutation testing score of 88 percent is not easy to achieve. But still, there are holes.

The most interesting mutants are the ones that appear on the green lines that indicate they were covered by tests. If a mutant was not covered by a test, it is not surprising that it survived and does not give any additional information compared to line coverage. But if a mutant was covered, you have something to investigate.



```
     /**
      * Fails if the multimap does not have the given size.
      */
     public void hasSize(int expectedSize) {
3      checkArgument(expectedSize >= 0, "expectedSize(%s) must be >= 0", expectedSize);
         1. hasSize : removed conditional - replaced equality check with false → SURVIVED
         2. hasSize : negated conditional → KILLED
2
1        failWithBadResults("has a size of", expectedSize, "is", actualSize);
       }
     }

     /**
      * Fails if the multimap does not contain the given key.
      */
     public void containsKey(@Nullable Object key) {
2      if (!getSubject().containsKey(key)) {
1        fail("contains key", key);
       }
     }
```

Figure 1. A report generated by pitest

For example, take a look at line 73 of PrimitiveIntArraySubject.java. Pitest created a mutant that has the following description:

```
removed call to com/google/common/truth/
PrimitiveIntArraySubject::failWithRawMessage
```

[This message has been wrapped due to width constraints. —*Ed.*] What this tells you is that pitest commented out the line of code that called this method.

As the name suggests, the purpose of failWithRawMessage is to throw a RuntimeException. Google Truth is an assertion library, so one of the core things that it does is throw an AssertionError when a condition is not met.

Let's take a look at the tests that cover this class. The following test looks like it is intended to test this functionality.

```
@Test
public void isNotEqualTo_FailSame() {
  try {
    int[] same = array(2, 3);
    assertThat(same).isNotEqualTo(same);
  } catch (AssertionError e) {
    assertThat(e)
      .hasMessage("<(int[]) [2, 3]>" +
                "unexpectedly equal to [2, 3].");
  }
}
```

Can you spot the mistake? It is a classic testing bug: the test checks the content of the assertion message but, if no exception is thrown, the test passes. Tests following this pattern normally include a call to fail(). Because the exception the Truth team expected is itself an AssertionError, the pattern they followed in other tests is to throw an Error.

```
@Test
public void isNotEqualTo_FailSame() {
  try {
    int[] same = array(2, 3);
    assertThat(same).isNotEqualTo(same);
    throw new Error("Expected to throw");
  } catch (AssertionError e) {
    assertThat(e)
      .hasMessage("<(int[]) [2, 3]>" +
                  "unexpectedly equal to [2, 3].");
  }
}
```

If this throw is added to the test, the mutant is killed.

What else can pitest find? There is a similar problem on line 121 of PrimitiveDoubleArraySubject.java. Again, pitest has removed a call to failWithRawMessage.

However, if you take a look at the test, it does throw an Error when no exception is thrown. So what's going on? This is an *equivalent mutant*. Let's examine this category of mutants a bit more.

Equivalent Mutants

Equivalent mutants are the other problem identified by the academic research that I referred to in the introduction.

Sometimes, if you make a change to some code, you don't actually change the behavior at all. The changed code is logically equivalent to the original code. In such cases, it is not possible to write a test that will fail for the mutant that doesn't also fail for the unmutated code. Unfortunately, it is impossible to automatically determine whether a surviving mutant is an equivalent mutant or just lacks an effective test case. This situation requires a human to examine the code. And that can take some time.

There is some research that suggests it takes about 15 minutes on average to determine if a mutation is equivalent.

So if you apply mutation testing at the end of a project and have hundreds of surviving mutants, you might need to spend days assessing the surviving ones to see whether they were equivalent.

This was seen as a major problem that must be overcome before mutation testing could be used in practice. However, much of the early research into mutation testing had an unstated built-in assumption. It assumed that mutation testing would be applied at the end of a development process as some sort of separate QA process. Modern development doesn't work like that.

The experience of people using pitest is that equivalent mutants are not a major problem. In fact, they can sometimes be helpful.

The most effective time to perform mutation tests on your code is when you write the code. If you do this, you will need to only assess a small number of surviving mutants at any one time, but, more importantly, you will be in a position to act on them. Assessing each surviving mutant takes far less than the suggested average of 15 minutes, because the code and the tests are fresh in your mind.

The only code you need to perform mutation testing on is code that you've just written or changed.

When a mutant in code you have just written survives, this will prompt you to do one of three things.

- If the mutant is not equivalent, you will most likely add a test.
- If the mutant is equivalent, you will often delete some code. One of the most common types of equivalent mutants is a mutation in code that does not need to be there.
- If the code is required, the equivalent mutation might prompt you to examine what the code is doing and the way it is implemented.

Line 121 of `PrimitiveDoubleArraySubject.java`, which you just examined, is an example of this last category. Let's take a look at the full method.

```
public void isNotEqualTo(Object expectedArray
                       , double tolerance) {
  double[] actual = getSubject();
  try {
    double[] expected = (double[]) expectedArray;
    if (actual == expected) {
      // the mutation is to the line below
      failWithRawMessage(
          "%s unexpectedly equal to %s."
           , getDisplaySubject()
           , Doubles.asList(expected));
    }
    if (expected.length != actual.length) {
      return; //Unequal-length arrays are not equal.
    }
    List<Integer> unequalIndices =
       new ArrayList<>();
    for (int i = 0; i < expected.length; i++) {
      if (!MathUtil.equals( actual[i]
                          , expected[i]
                          , tolerance)) {
        unequalIndices.add(i);
      }
    }
    if (unequalIndices.isEmpty()) {
      failWithRawMessage(
          "%s unexpectedly equal to %s."
          , getDisplaySubject()
          , Doubles.asList(expected));
    }
  } catch (ClassCastException ignored) {
    // Unequal since they are of different types.
  }
}
```

Pitest has mutated a method call that is conditionally executed after comparing two arrays with the == operator.

If the code does not throw an exception at this point, it will move on and perform a deep comparison of the arrays. If they are not equal, the code throws exactly the same exception as if the == had returned true.

So, this is a mutation in code that exists solely for performance reasons. Its purpose is to avoid performing a more expensive deep comparison. A large number of equivalent mutants fall into this category; the code is needed but relates to a concern that is not testable via unit tests.

The first question this raises is whether the behavior of this method should be the same when given the same array as it is when given two different arrays with the same contents.

My view is that it should not. If I am using an assertion library and I tell it that I expect two arrays not to be equal, and then I pass it the same array twice, I would find it useful for the message to tell me this, perhaps by adding "(in fact, it is the same array)" to the end of the failure message.

But perhaps I am wrong. Maybe the behavior is better the way it is. If the behavior remains the same, what can be done to make the equivalent mutation go away?

I don't like the isNotEqualTo method. It has two responsibilities. It is responsible for comparing arrays for equality and it is responsible for throwing exceptions when passed two equal arrays.

What happens if those two concerns are separated into different methods by doing something like this?

```
public void isNotEqualTo(Object expectedArray
                       , double tolerance) {
  double[] actual = getSubject();
  try {
    double[] expected = (double[]) expectedArray;
    if (areEqual(actual, expected, tolerance)) {
      failWithRawMessage(
          "%s unexpectedly equal to %s."
        , getDisplaySubject()
        , Doubles.asList(expected));
    }
  } catch (ClassCastException ignored) {
    // Unequal since they are of different types.
  }
}

private boolean areEqual(double[] actual
                       , double[] expected
                       , double tolerance) {
  if (actual == expected) return true;

  if (expected.length != actual.length)
   return false;

  return compareArrayContents(actual
                            , expected
                            , tolerance);
}
```

Now, the equivalent mutant goes away. The mutant has prompted me to refactor the code into something cleaner. What is more, I can now also use the new areEqual method to remove duplicate logic elsewhere in this class, thereby reducing the amount of code.

Unfortunately, not all equivalent mutants can be removed by re-expressing the code. If I uncomment the section of the configuration that enables pitest's stronger set of mutation operators and rerun the test, I'll get a mutant in the new areEqual method.

```
removed conditional -
replaced equality check with false
```

Pitest has changed the method to this:

```
private boolean areEqual(double[] actual
                       , double[] expected
                       , double tolerance) {
  if (false) return true; // mutated

  if (expected.length != actual.length)
      return false;

  return compareArrayContents(actual
                            , expected
                            , tolerance);
}
```

I can't refactor the equivalent mutant away without losing the performance optimization.

So not all equivalent mutants are helpful, but they are less common than the research suggests.

Pitest is designed to make equivalent mutants as unlikely as possible: using the default set of operators, many teams never encounter one. How many you see depends on the type of code you are writing and your coding style.

What About Really Big Projects?

None of the example projects I've talked about so far has been huge. Is it possible to use mutation testing on a really big project? Yes.

As I have discussed, by far the most effective way to use mutation testing is to run tests as you are developing code. When you use it in this way, project size doesn't matter. For a project such as Truth, it is simplest to mutate the entire project each time, but you don't need to do this.

The only code you need to perform mutation testing on is code that you've just written or changed. Even if your codebase contains millions of lines of code, it is unlikely that your code change will affect more than a handful of classes.

Pitest makes it easy to work in this way by integrating with version control systems. This functionality is currently available only when using the Maven plugin.

If you have correctly configured the standard Maven version control information in your POM file, you can analyze just your locally modified code using pitest's `scmMutationCoverage` goal.

This goal has been bound to the profile `pitest-local` in the Google Truth POM:

```
mvn -Ppitest-local test
```

If you haven't made any changes to the checked-out code, this goal will run the tests and then stop. If you have made changes, it will analyze only the changed files. Make a change now to try it out.

This approach gives you just the information you need: Is the code you're currently working on well tested? Pitest can also be set up so that a continuous integration (CI) server can analyze just the last commit.

But what if you want a complete picture of how good the tests are for a whole project?

Eventually, you will hit a limit for how large a project you can do mutation testing for unless you are willing to wait for many hours, but pitest does provide an experimental option to push that limit further.

Go back to the Google Truth project and run it with the following:

```
mvn -DwithHistory -Ppitest test
```

Nothing will seem very different from when you ran it before.

If you run that command again, however, it should finish in just a few seconds.

The withHistory flag tells pitest to store information about each run and use it to optimize the next run. If, for example, a class and the tests that cover it have not changed, there is no need to rerun the analysis on that class. There are many similar optimizations that can be performed using the run history.

This functionality is still in the early stages, but if it is used from the start of a project it should enable an entire codebase to be analyzed no matter how large it grows.

Conclusion

I hope I've convinced you that mutation testing is a powerful and practical technique. It helps build strong test suites and helps you to write cleaner code.

But I want to finish with a word of warning. Mutation testing does not guarantee that you will have good tests. It can only guarantee that you have strong tests. By strong, I mean tests that fail when important behavior of the code changes. But this is only half the picture. It is equally important that a test not fail when the behavior is left the same but details of the implementation are changed. </article>

Henry Coles (@0hjc) is a software engineer based in Edinburgh, Scotland, where he runs the local JUG. He has been writing software professionally for almost 20 years, most of it in Java. Coles has produced many open source tools including pitest and an open source book, *Java for Small Teams*.

learn more

Mutation testing on Wikipedia

FEATURED JAVA SPECIFICATION REQUEST

JSR 367: JSON Binding with JSON-B

The recently established Java API for JSON Processing (JSON-P) specification defines a standard API for parsing and generating JSON data. This standard was recently explained with detailed examples in our July/August issue (page 31). Although there are multiple JSON parser implementations, the benefit of JSON-P is that it will be bundled in the upcoming Java EE 8 release; in fact, it will be the defined standard for JSON parsing.

JSR 367 proposes to standardize a way to convert JSON into Java objects and vice versa. As proposed, JSON-B will leverage JSON-P and provide a conversion layer above it. The final definitive version of JSON-B will also be a part of the Java EE 8 release and, similarly to JSON-P, will be the defined standard binding layer for converting Java objects to and from JSON messages.

This JSR proposes a default mapping algorithm for converting existing Java classes to JSON. Those default mappings can then be customized through the use of Java annotations. JSON-B will be useful to other layers, such as JAX-RS.

Because the JSR is so far along in its finalization process, a PDF document of the specification released earlier this year is available for download. In addition, an elegant website with information on getting started and other resources is available. As promised in the article referenced earlier, *Java Magazine* will soon cover this emerging standard. But until then, these JSR resources should prove useful.

Implementing Design Patterns with Lambdas

Astute use of lambdas can greatly reduce the complexity of implementing standard coding patterns.

RAOUL-GABRIEL **URMA**, MARIO **FUSCO**, AND ALAN **MYCROFT**

New language features often make existing code patterns or idioms less popular. For example, the introduction of the for-each loop in Java 5 replaced many uses of explicit iterators because it's less error-prone and more concise. The introduction of the diamond operator, <>, in Java 7 reduced the use of explicit generics at instance creation (and slowly pushed Java programmers toward embracing type inference). In this article, we examine how lambdas can reduce the code needed to implement several programming patterns. To follow along, you'll need a basic familiarity with lambdas.

A specific class of patterns is called *design patterns*. They are a reusable blueprint, if you will, for a common problem when designing software. It's a bit like how construction engineers have a set of reusable solutions to construct bridges for specific scenarios (such as suspension bridge, arch bridge, and so on). For example, the visitor design pattern is a common solution for separating an algorithm from a structure on which it needs to operate. Another pattern, the singleton pattern, is a common solution to restrict the instantiation of a class to only one object.

Lambda expressions provide yet another new tool in the programmer's toolbox. They can provide alternative solutions to the problems the design patterns are tackling but often with less work and in a simpler way. Many existing object-oriented design patterns can be made redundant or written in a more concise way using lambda expressions. In this section, we explore design patterns:

- Strategy
- Template method
- Observer
- Factory

We show how lambda expressions can provide an alternative way to solve the same problem for which each of these design patterns is intended.

Strategy Pattern

The strategy pattern is a common solution for representing a family of algorithms and letting you choose among them at runtime. You can apply this pattern to a multitude of scenarios, such as validating an input with different criteria, using different ways of parsing, or formatting an input.

The strategy pattern consists of three parts, as illustrated in Figure 1.

These parts are:

- An interface to represent some algorithm (the interface `Strategy`)
- One or more concrete implementations of that interface to represent multiple algorithms (the concrete classes `ConcreteStrategyA` and `ConcreteStrategyB`)
- One or more clients that use the strategy objects

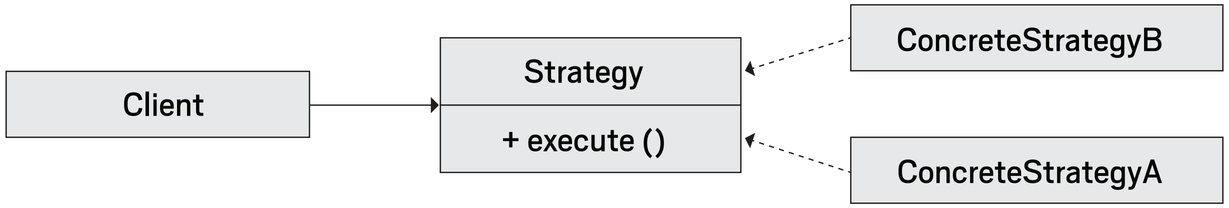


Figure 1. The strategy design pattern

Let's say you'd like to validate whether text input is properly formatted for different criteria (for example, it consists of only lowercase letters or is numeric). You start by defining an interface to validate the text (represented as a `String`):

```
public interface ValidationStrategy {
    boolean execute(String s);
}
```

Second, you define one or more implementation(s) of that interface:

```
public class IsAllLowerCase
  implements ValidationStrategy {
    public boolean execute(String s){
        return s.matches("[a-z]+");
    }
}

public class IsNumeric
  implements ValidationStrategy {
    public boolean execute(String s){
        return s.matches("\\d+");
    }
}
```

You can then use these different validation strategies in your program:

```
public class Validator{
   private final ValidationStrategy strategy;

   public Validator(ValidationStrategy v){
       this.strategy = v;
   }

   public boolean validate(String s){
       return strategy.execute(s);
   }
}
```

Then with this code, the first example returns `false`, the second one `true`:

```
Validator v1 =
    new Validator(new IsNumeric());
System.out.println(v1.validate("aaaa"));

Validator v2 =
    new Validator(new IsAllLowerCase());
System.out.println(v2.validate("bbbb"));
```

You should recognize that `ValidationStrategy` is a functional interface. This means that instead of declaring new classes to implement different strategies, you can pass lambda expressions directly, which are more concise:

```
// with lambdas
Validator v3 =
  new Validator((String s) ->
       s.matches("\\d+"));
System.out.println(v3.validate("aaaa"));

Validator v4 =
  new Validator((String s) ->
      s.matches("[a-z]+"));
System.out.println(v4.validate("bbbb"));
```

As you can see, lambda expressions remove the boilerplate code inherent in the strategy design pattern. If you think about it, lambda expressions encapsulate a piece of code (or strategy), which is what the strategy design pattern was created for, so we recommend that you use lambda expressions instead for similar problems.

Template Method Pattern

The template method design pattern is a common solution when you need to represent the outline of an algorithm and have the additional flexibility to change certain parts of it. In other words, the template method pattern is useful when you find yourself in a situation such as "I'd love to use this algorithm but I need to change a few lines so it does what I want."

Let's look at an example of how this pattern works. Say you need to write a simple online banking application. Users typically enter a customer ID, and then the application fetches the customer's details from the bank database and finally does something to make the customer happy. Different online banking applications for different banking branches may have different ways of making a customer happy (for example, adding a bonus on the account or just sending less paperwork). You can write the following abstract class to represent the online banking application:

```
abstract class OnlineBanking {

    public void processCustomer(int id){
        Customer c = Database.getCustomerWithId(id);
        makeCustomerHappy(c);
    }
    abstract void makeCustomerHappy(Customer c);
}
```

The processCustomer method provides a sketch for the online banking algorithm: fetch the customer given his or her ID and then make the customer happy. Different branches can now provide different implementations of the method makeCustomerHappy by subclassing the OnlineBanking class.

You can tackle the same problem (creating an outline of an algorithm and letting implementers plug in some parts) using your favorite lambdas. The different components of the algorithms you want to plug in can be represented by lambda expressions or method references.

Here, we introduce a second argument to the method processCustomer of type Consumer<Customer> because it matches the signature of the method makeCustomerHappy defined earlier:

```
public void processCustomer(
  int id, Consumer<Customer> makeCustomerHappy){
    Customer c = Database.getCustomerWithId(id);
    makeCustomerHappy.accept(c);
}
```

You can now plug in different behaviors directly without subclassing the OnlineBanking class by passing lambda expressions:

```
new OnlineBankingLambda().processCustomer(1337,
 (Customer c) -> System.out.println(
    "Hello " + c.getName());
```

This is another example of how lambda expressions can help you remove the boilerplate inherent in design patterns!

Observer Pattern

The observer design pattern is a common solution when an object (called the *subject*) needs to automatically notify a list of other objects (called *observers*) that some event has occurred (for example, a state change). You typically come across this pattern when working with GUI applications. You register a set of observers on a GUI component such as a button. If the button is clicked, the observers are notified and can execute a specific action. But the observer pattern isn't limited to GUIs. For example, the observer design pattern is also suitable in a situation where several traders (observers) might wish to react to the change of price of a stock (subject). Figure 2 illustrates the UML diagram of the observer pattern.

Let's write some code to see how the observer pattern is useful in practice. You'll design and implement a customized notification system for an application like Twitter. The concept is simple: several newspaper agencies (*The New York Times*, *The Guardian*, and *Le Monde*) are subscribed to a feed of news tweets and may want to receive a notification if a tweet contains a particular keyword.

First, you need an Observer interface that groups the different observers. It has just one method called notify that will be called by the subject (Feed) when a new tweet is available:

```
interface Observer {
    void notify(String tweet);
}
```

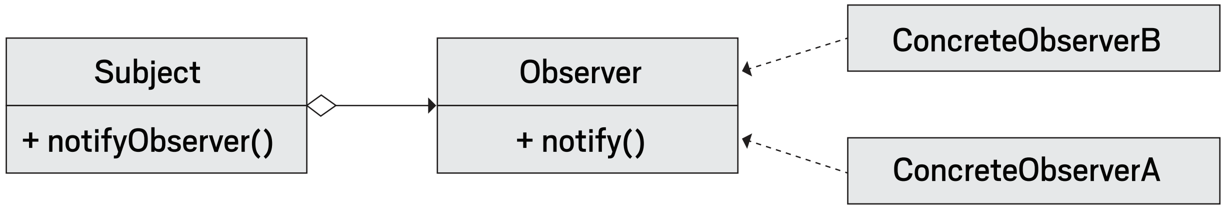


Figure 2. The observer design pattern

You can now declare different observers (here, the three newspapers) that produce a different action for each different keyword contained in a tweet:

```
class NYTimes implements Observer{
  public void notify(String tweet) {
    if(tweet != null && tweet.contains("money")){
       System.out.println(
         "Breaking news in NY! " + tweet);
        }
    }
}
class Guardian implements Observer{
  public void notify(String tweet) {
    if(tweet != null && tweet.contains("queen")){
       System.out.println(
         "Yet more news in London... " + tweet);
        }
    }
}
class LeMonde implements Observer{
  public void notify(String tweet) {
    if(tweet != null && tweet.contains("wine")){
       System.out.println(
         "Today cheese, wine, and news! " + tweet);
        }
    }
}
```

You're still missing the crucial part: the subject! Let's define an interface for it:

```
interface Subject{
    void registerObserver(Observer o);
    void notifyObservers(String tweet);
}
```

The subject can register a new observer using the registerObserver method and notify his observers of a tweet with the notifyObservers method. Let's go ahead and implement the Feed class:

```
class Feed implements Subject{

    private final List<Observer> observers =
        new ArrayList<>();

    public void registerObserver(Observer o) {
        this.observers.add(o);
    }

    public void notifyObservers(String tweet) {
        observers.forEach(o -> o.notify(tweet));
    }
}
```

It's a pretty straightforward implementation: the feed keeps an internal list of observers that it can then notify when a tweet arrives. You can now create a demo application to wire together the subject and observers:

```
Feed f = new Feed();
f.registerObserver(new NYTimes());
f.registerObserver(new Guardian());
f.registerObserver(new LeMonde());
f.notifyObservers(
  "The queen said her favourite book " +
  "is Java 8 in Action!");
```

Unsurprisingly, *The Guardian* will pick up this tweet!

You might be wondering how lambda expressions are useful with the observer design pattern. Notice that the different classes implementing the Observer interface are all providing implementation for a single method: notify. They're all just wrapping a piece of behavior to execute when a tweet arrives. Lambda expressions are designed specifically to remove that boilerplate. Instead of instantiating three observer objects explicitly, you can pass a lambda expression directly to represent the behavior to execute:

```
f.registerObserver((String tweet) -> {
  if(tweet != null && tweet.contains("money")){
    System.out.println(
      "Breaking news in NY! " + tweet);
  }
});

f.registerObserver((String tweet) -> {
  if(tweet != null && tweet.contains("queen")){
    System.out.println(
      "Yet more news from London... " + tweet);
  }
});
```

Should you use lambda expressions all the time? The answer is no. In the example we described, lambda expressions work great because the behavior to execute is simple, so they're helpful to remove boilerplate code. But the observers may be more complex: they could have state, define several methods, and the like. In those situations, you should stick with classes.

Factory Pattern

The factory design pattern lets you create objects without exposing the instantiation logic to the client. For example, let's say you're working for a bank and it needs a way of creating different financial products: loans, bonds, stocks, and so on.

Typically you'd create a Factory class with a method that's responsible for the creation of different objects, as shown here:

```
public class ProductFactory {
  public static Product createProduct(String name){
    switch(name){
      case "loan": return new Loan();
      case "stock": return new Stock();
      case "bond": return new Bond();
      default: throw new RuntimeException(
                "No such product " + name);
    }
  }
}
```

Here, Loan, Stock, and Bond are all subtypes of Product. The createProduct method could have additional logic to configure each created product. But the benefit is that you can now create these objects without exposing the constructor and the configuration to the client, which makes the creation of products simpler for the client:

```
Product p = ProductFactory.createProduct("loan");
```

In lambda expressions, you can refer to constructors just like you refer to methods, by using method references. For example, here's how to refer to the Loan constructor:

```
Supplier<Product> loanSupplier = Loan::new;
Loan loan = loanSupplier.get();
```

Using this technique, you could rewrite the previous code by creating a Map that maps a product name to its constructor:

```
final static Map<String, Supplier<Product>> map =
     new HashMap<>();

static {
    map.put("loan", Loan::new);
    map.put("stock", Stock::new);
    map.put("bond", Bond::new);
}
```

You can now use this Map to instantiate different products, just as you did with the factory design pattern:

```
public static Product createProduct(String name){
    Supplier<Product> p = map.get(name);
    if(p != null) return p.get();
    throw new IllegalArgumentException(
        "No such product " + name);
}
```

This is quite a neat way to make use of the Java 8 features to achieve the same intent as the factory pattern. But this technique doesn't scale very well if the factory method createProduct needs to take multiple arguments to pass on to the product constructors. You'd have to provide a different functional interface than a simple Supplier.

These examples make clear that lambdas can be used in many situations in which you might not normally think of applying them. Getting in the habit of using lambdas, however, will make your code shorter, clearer, and easier to write. </article>

This article is adapted from an excerpt of the book Java 8 in Action by Raoul-Gabriel Urma, Mario Fusco, and Alan Mycroft. Used with permission.

learn more

The original "Gang of Four" book on design patterns

MARTIJN VERBURG

Contribute to Java by Adopting a JSR

How you can build your developer skills by contributing to Java standards

PHOTOGRAPH BY BOB ADLER

The Adopt-a-JSR program was designed by Oracle "to encourage JUG members, individuals, corporations and other organizations to get involved in Java Specification Requests (JSRs)." It lets Java developers be involved in the development of the Java platform, via the standards body. In this article, I explain what Adopt-a-JSR is, the goals of the program, and how you can get involved.

What Is Adopt-a-JSR?

Adopt-a-JSR is focused on getting Java developers involved with emerging Java standards as early as possible. Individual developers, Java user groups, and organizations are all eligible and welcome to join.

The Java standards covered by JSRs determine what goes into Java SE, Java EE, and Java ME. For example, Project Jigsaw, which provides a modularization platform for Java SE, is defined in JSR 376. For Java EE, the Servlet 4.0 specification is determined in JSR 369.

Why Participate in Adopt-a-JSR?

The main reason for this program is to ensure that the emerging standards for the Java ecosystem are technically sound, relevant, usable, accessible, and free from intellectual property entanglements. With an estimated 10 million developers using Java, this kind of attention to emerging standards is important.

This program enhances a Java developer's technical, social, and strategic skills and credentials. Many developers involved in Adopt-a-JSR have gone on to become authors, speakers, and leaders in various projects and organizations.

Regarding the work itself, Java developers who adopt a JSR are often involved in activities such as

- Requirements gathering
- Specification design
- Reference implementation work
- Technology compatibility kit work

If you'd like to help Java continue to be the leading language and platform choice for software engineers, then this is a great place to start.

Many potential contributors are reluctant to participate in a JSR because they're not experts. But JSRs should be seen more as special interest groups (SIGs) on a particular topic, in which the members fulfill a wide variety of different needs. Many of the tasks listed above require you only to have an interest in the topic and the willingness to spend some time. They're not reserved for domain experts.

Get Started

To get started, join the community at Adopt-a-JSR. There are links on that portal for signing up for the IRC channel, mailing lists, and so forth. The next step is to pick an active JSR that you and your colleagues are personally interested in and passionate about. You can then post your interest to the Adopt-a-JSR mailing list, where you will get advice on how to join the official JSR you're interested in and how best to contribute.

When you join the official JSR mailing list and project, you will typically contact the specification leads and the expert group and ask them for tasks that need to be worked on. These could be something as small as helping out with social media awareness or as involved as writing JUnit tests for the technology compatibility kit. It will vary depending on what stage the JSR is in at the time.

If you've wondered how Java SE, Java EE, and Java ME are defined and wish you were able to help, then **Adopt-a-JSR is the place to start.**

Similar to participating in any good open source project, you should always try to coordinate your efforts with others in the program who have adopted the JSR as well as the specification leads and the expert group.

Building an online presence for your work is important as well. It's helpful to have a wiki page for your work on the JSR. (The London JUG's page is a good example.) Your wiki page should explain what you're aiming to do and link to related material and help for other volunteers. It also helps to spread visibility within your local JUG or organization through mailing lists and social media.

The Adopt-a-JSR community can help you with this, as it offers hosting for wiki pages, a GitHub hosting service for code work, and more.

Once you are up and running, there are a lot of ongoing tasks that you can do to make a meaningful contribution. Here's a sample list:

- Evangelize the JSR through social media, blogging, and talks
- Arrange "hack days" and meetups to work with, or test out, the JSR
- Help triage an issue in the issue trackers and constructively contribute to discussions on the mailing list
- Help build the reference implementation or technical compatibility kit.

See the Adopt-a-JSR page for more details on how you can get involved, including some Keynote and PowerPoint presentations you can give to your local JUG or organization.

Conclusion

If you've wondered how Java SE, Java EE, and Java ME are defined and wish you were able to help, then Adopt-a-JSR is the place to start. It's great for your career and will help millions of your fellow Java developers around the world. In the next issue of *Java Magazine*, I will have a similar article on contributing to OpenJDK via the Adopt OpenJDK program, which has a specific focus on Java SE. </article>

Martijn Verburg is a Java Champion and the cofounder and CEO of jClarity, a startup focused on the next generation of lightweight, intelligent performance analysis products. He is a coauthor of *The Well-Grounded Java Developer* (Manning, 2012) with Ben Evans, and is deeply involved in getting Java developers to participate in the development of the Java ecosystem.

learn more

An overview of the JSR universe

A time line showing the status of currently active JSRs

SHAY SHMELTZER

Getting Started with Agile Java Development in the Cloud

Avoid endless setup with Oracle Developer Cloud Service's integrated Git, build, deploy, code review, and project management tools.

Many development teams are looking for a development process that will accelerate delivery of applications and new features. Some of these teams are adopting an agile development methodology to help them achieve these goals. Much has already been written about how to bring agility to teams, so in this article I focus on using tools that help manage and control both code and the development team in the cloud as an integral part of adopting agility.

As your team moves to agile development, you'll find there is a vast array of tools that help with implementing agility. These tools include utilities that help manage a team's work, including issue trackers, agile management dashboards and reports, live team activity streams, and wikis. There are also tools to help manage the code lifecycle through version management, build servers, continuous integration engines, and deployment platforms.

One of the challenges for an organization moving into agile development is the need to get all of these utilities provisioned and integrated in a way that delivers a cohesive development and management platform. Add in the cost and time involved in maintaining the servers and software, and you have a bottleneck to agile adoption.

Luckily, cloud-based platforms are emerging that help with these challenges by easily provisioning integrated platforms. In this article, I focus on Oracle Developer Cloud Service and how it can facilitate adoption of agile development while cutting the cost and time associated with setting and integrating these tools.

Getting Started

Oracle Developer Cloud Service is included in the free trial and licensed versions of many Oracle cloud platform services such as Oracle Java Cloud Service (used for deploying Java EE apps) and Oracle Application Container Cloud (used for running Java SE loads and Node.JS apps). If you want to follow along with this article, get a trial of either of those services.

Provisioning a Platform

When you first log in to Oracle Developer Cloud Service, you'll see a list of the projects you have access to as a team member as well as projects that are marked as public in your cloud instance.

A project is the base environment for the team. It includes an issue-tracking system and a wiki, one or more Git repositories, a peer code-review system, and build and deployment processes.

To provision a project, click the project creation button and use the simple three-step wizard, with which you specify a name for your project, a template, and a wiki markup choice. The project can be private or public.

Once you finish with the wizard, Oracle Developer Cloud Service provisions your environment in about one minute. This is one of the great things about a cloud environment—a setup process that used to take days if not weeks and involved setting up servers, installing software, and hooking up the various components can make those resources available almost immediately.

Once your project is provisioned, you'll have a full environment that includes

- Git repository
- Issue-tracking system
- Wiki
- Code review system
- Continuous integration server and Maven repository
- Team activity stream

The next step is to add the rest of the team members to the project and specify their roles. Members in a team can include developers, quality assurance (QA), documentation, and operations people, as well as any stakeholder who is part of the agile team.

As an administrator, you can also further configure your project with specific values for various lists in the issue-tracking system and agile planning steps, as well as creating and mapping code repositories. You can also integrate with external systems via web hooks.

You can use Gradle, Maven, or Ant for Java-based builds, and if you are also doing JavaScript/Node.JS coding, then you can add npm, Gulp, Grunt, and Bower steps to your build process.

Managing Code

You might want to start by uploading your current code into the Git repository. You can use regular Git command lines or any IDE with Git integration to connect to the Git repository using either HTTPS or SSH.

Once your code has been uploaded, you can browse it from the Code tab in your web browser.

You can now continue using your preferred Git interaction pattern to manage your code, including creating branches, tagging, and so on. Note that the Code tab also enables you to view changes in your code in a visual way, comparing revisions.

Conducting Code Reviews

Are you ready to merge changes from one branch into another? You can submit a merge request and specify individual members of your team that should review your code before it gets into the application.

These members will get an email notifying them of the review request, and they can use a browser to see the changes you made and comment on specific lines of code (see **Figure 1**). Your back-and-forth discussion can be tracked online by other reviewers, too. Once your branch is ready, you can use the browser merge button to merge your code into your target branch.

Automating Builds

The Build tab in Oracle Developer Cloud Service lets you define various build steps for your project. You can use Gradle, Maven, or Ant for Java-based builds, and if you are also doing JavaScript/Node.JS coding, then you can add npm, Gulp, Grunt, and Bower steps to your build process.

Execution, orchestration, and scheduling of builds are done with a cloud-optimized version of Hudson. You can define builds to run at specific times, or based on the results of another build, or even automatically when a change is committed into a specific branch of your Git repository.

As part of your build, you can run tests of the back end (using JUnit), the front end (using Selenium), and the code quality of your application (using FindBugs).

Oracle Developer Cloud Service integration of popular build and testing frameworks makes it simple for your development team to adopt it as a platform for managing your build automation, leveraging the knowledge you already possess in these frameworks.

Streamline Deployment

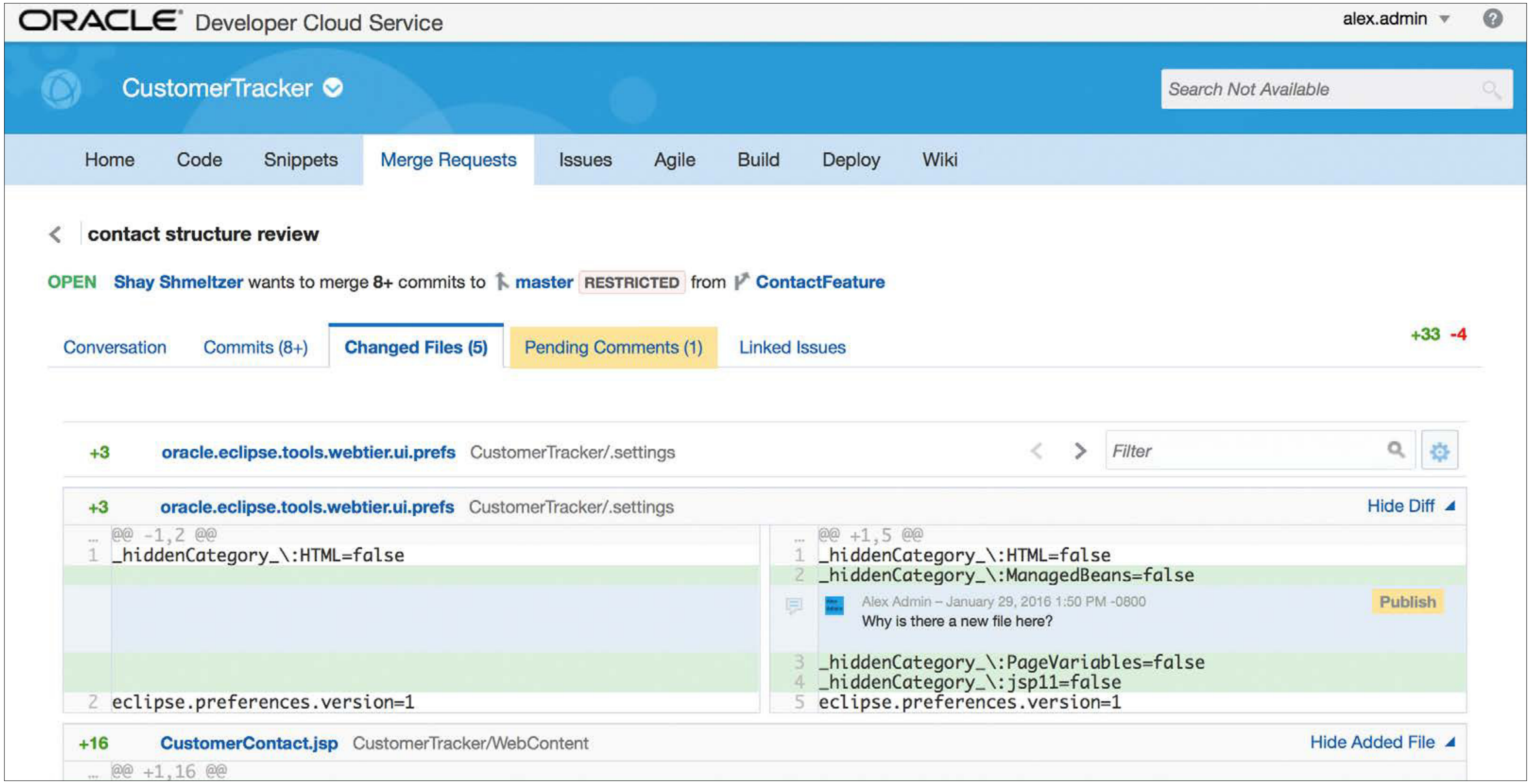
The Deploy tab in Oracle Developer Cloud Service completes the code lifecycle by allowing you to push your code into the runtime environment. Whether you're using Oracle Java Cloud Service (think of it as WebLogic in the cloud) or Oracle Application Container Cloud (on which you can run your Java SE apps and servers), you can define deployment profiles that will deploy your code into the runtime environment.

You can tie the deployment step to the success of specific builds, enabling you to automate the full process from code check-in through compile, package, and test all the way to having a running version of the code in minutes.

Deployment can be done to multiple servers so, for example, you can manage deployment to a development instance and a QA instance separately from the deployment to the production instance.

Tracking Your To-Do List

Oracle Developer Cloud Service also provides a platform for managing your tasks/issues and your teamwork. The Issues



contact structure review

OPEN Shay Shmeltzer wants to merge 8+ commits to master RESTRICTED from ContactFeature

+3 oracle.eclipse.tools.webtier.ui.prefs CustomerTracker/.settings

+3 oracle.eclipse.tools.webtier.ui.prefs CustomerTracker/.settings

```
@@ -1,2 @@
_hiddenCategory_\:HTML=false
eclipse.preferences.version=1
```

```
@@ +1,5 @@
_hiddenCategory_\:HTML=false
_hiddenCategory_\:ManagedBeans=false
_hiddenCategory_\:PageVariables=false
_hiddenCategory_\:jsp11=false
eclipse.preferences.version=1
```

Alex Admin – January 29, 2016 1:50 PM -0800
Why is there a new file here?

+16 CustomerContact.jsp CustomerTracker/WebContent

```
@@ +1,16 @@
```

Figure 1. Inline commenting on code changes as part of code review

tab offers an easy way for team members to report new issues such as defects, features, and tasks that need to be addressed. Issues can be associated with dynamic lists of application features, releases, sprints, and so forth—all customizable by the project administrator.

You can create subissues to track more-complex tasks with various dependencies. Each task can also be associated with an estimate of time it will take to complete or with an agile point-ranking score. You can also assign specific tasks to specific team members.

Managing Agile Sprints

Oracle Developer Cloud Service can help you manage your development sprints. Define a new sprint and decide which tasks from your backlog are going to be addressed in the sprint with a simple drag and drop.

Then you can create boards that will show you live information about the status of each task and the load on each team member as the sprint progresses. Using the Reports tab, you'll be able to see your progress trend toward the completion of the tasks in the current sprints.

And based on your results in previous sprints, the service will alert you if you're trying to address too many issues that require too much time (or have a high agile point count). This helps prevent you from missing your deadlines.

Team Communication

Team communication is crucial for a successful agile development process, and Oracle Developer Cloud Service aims to help here, too. Wikis allow team members to share knowledge, including design documents, coding practices, specs, and any other information that can help team members understand and complete their tasks.

To keep you up to date with what is happening in the project, the activity stream on the projects home page lists events as they occur. Whether a new Git branch got a commit, a build failed, or a new task was updated, the activity stream allows you to keep current with what's going on with the rest of the team.

IDE Integration

So far, I have focused on the web-based interfaces for interacting with Oracle Developer Cloud Service. But given that developers spend much of their day inside their development environment, Oracle has added integration features specifically for the service into NetBeans, Eclipse, and JDeveloper.

From all three IDEs, developers are able to browse Oracle's cloud services to find their projects and access their Git repositories. In addition, developers can see the tasks in the system, including the specific tasks that are assigned to them. They can open the tasks and update them from the IDE as well.

One benefit of this integration is that code changes that are being committed to the Git repository can be associated with specific tasks that developers are working on, creating a deeper connection between the issue-tracking system and the code revision system.

Conclusion

Oracle Developer Cloud Service integrates software utilities that will help teams manage both their software and their team assignments. With nothing to install, fast provisioning, and deep IDE integration, Oracle Developer Cloud Service removes barriers to adopting agile development, resulting in faster delivery of better applications. </article>

Shay Shmeltzer (@jdevshay) is the director of product management for Oracle Cloud Development Tools and Frameworks.

SIMON ROBERTS

Quiz Yourself

Would you ever use a finally clause with a try-with-resources? Explore this and other subtle questions from an author of the Java certification tests.

Once again, I've composed more problems that simulate questions from the 1Z0-809 Programmer II exam, which is the certification test for developers who have been certified at a basic level of Java 8 programming knowledge and now are looking to demonstrate more-advanced expertise. [Readers looking for basic instruction should consult the New to Java column, which appears in every issue. —*Ed.*] As usual, these questions can require careful deduction to obtain the right answer.

Question 1. Given the following code:

```
ResourceBundle properties = ResourceBundle.getBundle(
  "scratch.ConfigData", Locale.FRANCE); // line n1

System.out.println(
  properties.getString("help")); // line n2
```

And a directory and text file `scratch/ConfigData.properties` located on the `CLASSPATH` of the running program, which contains the following text:

```
file=File
edit=Edit
help=Help
```

What is the result? Choose one.

a. `help`
b. `Help`
c. `Aide`
d. An exception at line n1
e. An exception at line n2

Question 2. Given this code:

```
Collection<String> coll = new ArrayList<>();
coll.add("Fred"); coll.add("Jim"); coll.add("Sheila");
System.out.println("coll is " + coll);
coll.remove(0); // line n1
System.out.println("coll is " + coll);
```

What is the result? Choose one.

a. `coll is [Fred, Jim, Sheila]`
`coll is [Jim, Sheila]`
b. `coll is [Fred, Jim, Sheila]`
`coll is [Fred, Jim, Sheila]`
c. Compilation fails at line n1.
d. An exception is thrown at line n1.

Question 3. Given that the current working directory of the program is empty, and given this code fragment:

```
Path p = Paths.get("a", "b", "cee"); // line n1
System.out.println(p.endsWith(Paths.get("b", "cee")));
System.out.println(p.endsWith(Paths.get("ee")));
```

What is the result? Choose one.

a. `true`
`true`

b. true
false

c. false
true

d. false
false

e. An exception at line n1

Question 4. Given the following code:

```
class MyResource implements AutoCloseable {
    private String name;
    public MyResource(String name) {
        this.name = name;
    }

    @Override
    public void close() {
        System.out.println(name);
    }
}

public class ResourceHog {
    public static void main(String[] args) {
        try (MyResource one = new MyResource("one");
             MyResource two = new MyResource("two")) {
        } finally {
            System.out.println("three");
        }
    }
}
```

What is the result? Choose one.

a. one
two
three

b. three
two
one

c. three
one
two

d. two
one
three

e. The words one, two, and three are all output, but the order is not determined.

Question 5. Given this code:

```
class MyJobException extends Exception {}

class Job implements Callable<String> {
  public String call() throws MyJobException { // line n1
        return "Done";
    }
}

public class Jobs {
    public static void main(String[] args) {
        ExecutorService es =
            Executors.newFixedThreadPool(3);
        Future<String> handle = es.submit(new Job());
        try {
            String rv = handle.get();
        } // line n2
        catch (InterruptedException ex) {
            ex.printStackTrace();
        } // line n3
    }
}
```

Which is true? Choose one.

- **a.** The code compiles successfully.
- **b.** The code compiles if the `throws` clause at line n1 is deleted.
- **c.** The code compiles if `catch (Exception e){}` is added immediately after line n2.
- **d.** The code compiles if `catch (MyJobException e){}` is added immediately after line n3.
- **e.** The code compiles if `catch (Exception e){}` is added immediately after line n3.

Question 1. The correct answer is option B. This question is really asking what happens when you try to access a resource bundle for a locale, but there is no file defining values for that particular locale.

When you deploy an internationalized application, you try to provide resource files for all the locales in which you expect your software to be used, and perhaps you even offer updates at intervals as you find customers in new locations that you didn't originally support. However, if the program were to throw an exception when it is used in an unexpected locale, that would be pretty harsh; so it's a fair guess that throwing an exception isn't going to be a good behavior. Indeed, there's no checked exception when you try to access the file in the first place (at line n1) or when you look up the particular value (at line n2). On that basis, both option D and option E are incorrect.

If no exception is thrown, some text must result. The three remaining options amount to saying "Java can translate English to French and come up with *Aide*," "Java uses the key instead of a value if the value isn't found," and "Java uses a less perfectly appropriate resource file." The actual truth in this situation is the third of these: Java uses the best-fitting resource file it can find, even if that file is the default file.

Here, the only file that exists is the default resource file, but Java takes several tries before settling on this. A locale is made of several parts. I'll simplify the discussion by considering two main parts: the language and the region. In this case, `Locale.FRANCE` refers to the French language as spoken in France, which is coded as `fr_FR`. Given this data, the `ResourceBundle` mechanism tries to find these files in order: `ConfigData_fr_FR.properties` and `ConfigData_fr.properties`.

Notice that the filenames are constructed from three parts: the base bundle name (`ConfigData`, here), then the locale-derived part, and then the extension (`.properties`). Strictly speaking, several more local variations might be checked with more-specific information, but the concept is illustrated well enough here. If the search has still failed (as it will in this example), the search is repeated using the system's default locale (which might or might not be `fr_FR`). In this case, you don't know whether there's a different default locale, and it doesn't matter anyway, because only one file exists. So the second search fails, too. After that, the system falls back to searching for the default file, `ConfigData.properties`.

The effect of this is that the search finds the best-available match. If you had a file dedicated to French speakers in France, it would be found. If that's missing, but there's a generalized French-language file, that would be found. Finally, as is the case with this question, the system falls back to the nonspecific version, which, in this case, contains English. By the way, this nonspecific version has an official name, which is *base bundle*.

There is a failure mode that's broadly consistent with option E if all potential resource files, including the base bundle, are missing. In this situation, the getBundle method throws a MissingResourceException (which is an unchecked exception, so no code has to admit this possibility). Of course, that's not relevant to this question because the base bundle is found.

By the way, the process is explained in more detail in the API documentation.

This behavior is really quite useful. If you had a single French file, ConfigData_fr.properties, it would probably serve quite well for any French speakers, whether they were French or Canadian, or they were from any of nearly 30 countries in Africa or about a dozen others around the world.

It's interesting to note, of course, that Java has a nontrivial multilingual capability. While it generally does not attempt to translate text between languages, it knows the names of months and days of the week in many languages, so date-type objects are converted into locale-appropriate text without help from the programmer.

Question 2. The correct answer is option B. This is one of those questions that investigates a situation where misunderstanding can easily occur—with potentially difficult-to-debug consequences. This question might be too hard for quizzes, but it's a truly enlightening problem, so I'll share it with you nonetheless.

So, what's going on? Because the correct answer is option B, it appears that the code compiles and runs, but it does not modify the data in the collection. The behavior hinges on the method at line n1. If you look at this in an IDE, you'll see that the argument to the remove method is an Object, not an int. The remove(Object) method is defined on Collection, and it removes an object that matches the argument. But this collection contains only three strings: "Fred", "Jim", and "Sheila". Consequently, no change is made to the collection, and the answer makes sense.

I think that this leaves a couple of curiosities remaining. First, isn't there a remove(int) method defined for a List? And, doesn't that method remove the item at the given index position? Well, yes, both of those are true statements. However, the relationship between remove(int) and remove(Object) is one of method overloading, not method overriding, which means that the compiler decides which method to invoke, rather than the object that's the target of the invocation making the decision at runtime.

In this example, the collection object is actually a List, but the code refers to it as a Collection, and Collection does *not* have the remove(int) method (collections are not intrinsically ordered, so positional indexes are not meaningful). Because of this, the compile-time decision is made to invoke the remove(Object) method.

As a point of interest, if you simply change the declared type of the variable coll from Collection to List, "Fred" is removed from the list and the output does indeed change from what's shown for option B to what's shown for option A.

But at this point, you might reasonably ask why the compiler allows the call to remove on a Collection<String> to accept an Integer argument. Doesn't the generics mechanism restrict the argument to a String, and wouldn't that result in a compilation failure (option C)? If you check the documentation for this method, it does not actually take a generic argument; it takes Object. The same is true of the contains method, too. In effect, this allows you to say "if the code contains this item, remove it" while referring to an item that cannot possibly be present because it's the wrong type. This seemingly useless behavior allows for backward compatibility with code written prior to the advent of generics. Similarly, you can ask whether a collection of automobiles contains an apple, and while the question is valid, you just get the answer "no."

It's interesting to note that many IDEs and style guides (including the CERT Secure Java Coding recommendations) warn against calling these methods with generically inappropriate arguments, and you can probably see why now. Also, the return value of these `remove` methods can be used to determine whether any change was made, and style guides often advise against ignoring such return values.

From one perspective, the difference between inheritance and overloading is simple enough, but sometimes it takes on a trickier consequence—which makes a great question and discussion, partly because you might have a horrible time trying to debug the code when a manager is yelling at you to "Fix it now!"

Question 3. The correct answer is option B. The first observation is that the `Path` mechanism has no problem referring to files and directories that don't exist. That's actually quite important, because if it were unable to do so, you'd have a problem trying to create a new file. On that basis, you should be ready to eliminate option E, which seems to be hinting that the reference to the path a/b/cee, which doesn't exist, would cause a problem. (Note that the question states that the current working directory is empty.)

So, the remainder of the question revolves around two questions. First is whether the `Path.endsWith` method traverses directories (if it does, you would expect the first line of output to be `true`). Second is whether it examines whole segment names or is willing to match just the tail end of a segment name, such as deciding that `a/b/cee` ends with `ee`.

It's a rather boring matter of rote learning that, indeed, the method does match across directories, but it does not match partial segment names. Hence, the first output line is true, and the second is false—making option B the correct answer.

Question 4. The correct answer is option D. This question addresses the behavior of the try-with-resources mechanism and how it interacts with the `AutoCloseable` resources that it manages. Two different behaviors are addressed: 1) the timing relationship between the automatic closure and any explicitly coded `finally` block, and 2) the timing relationship among multiple resources all opened in the parameter list of a single `try` statement.

Much of the time when you use try-with-resources, there's no need for a `finally` block; the auto-generated code created by the compiler normally closes anything that needs to be closed. However, there are some resource-like things that do not implement `AutoCloseable`, and those would likely need a `finally` block to release them. The examples that spring to mind for me are the API-level lock utilities that exist in the `java.util.concurrent` package. Therefore, it is possible to have both try-with-resources and a `finally` block in well-thought-out code that isn't simply illustrating an academic talking point.

Of course, the relationship between any resource being released in a `finally` block and the resources being managed by the try-with-resources mechanism is pretty tenuous. The lexical scope of the managed resources is necessarily limited to the `try` statement and the block that immediately follows it. Therefore, those resources are generally invisible to the `finally` block. In contrast, anything visible in the `finally` block must be scoped "outside" of the whole try/catch/finally construction.

Nevertheless, Java specifies that the `finally` block is executed after the management of the `AutoCloseable` resources. *Java Language Specification* section 14.20.3.2, "Extended try-with-resources," states that "all resources will have been closed (or attempted to be closed) by the time the `finally` block is executed." This rules out option B and option C, which assert that the word `three` is printed first.

Also, section 14.20.3 (the overarching section containing the subsection mentioned above) notes that variables are "closed automatically, in the reverse order from which they were initialized." Consequently, you can be sure that the two resources will be closed such that two is output first and one comes after.

Notice that these specification statements are explicit; they don't leave the order of the operations uncertain in any way. The order has been pinned down as two, one, three, so the answer is not option E, but rather option D.

If you came across this in an exam, you might consider that it's a lot of code to understand for a sub-two-minute answer. However, notice that the options offered include only the ordering of output, so you can safely ignore minor syntactic details. Instead, you need to identify only that the difference between the various options amounts to "the order of closing is defined versus undefined" and then decide between "AutoCloseable first, and then finally" and its inverse and between "auto-close in reverse order from opening" and its inverse.

Question 5. The correct answer is option E. This question investigates the relationship between the abnormal termination (otherwise known as "exception behavior") of a Callable and the get method of the Future that connects to that Callable.

The general ExecutorService mechanism allows you to submit jobs, represented either as Runnable or Callable objects, to a pool of worker threads. The run method of a Runnable is declared to return void, and cannot throw any checked exceptions. In contrast, Callable is a generic interface, and the call method that it defines returns whatever the generic type might be. Also, call in the base interface is declared to throw Exception, permitting implementations to throw any exception they like. This means that option B, which effectively asserts that the call method may not throw an exception, is false.

Option C can immediately be rejected based on regular Java behavior. If a series of catch blocks catch exception types that are related by class hierarchy, the more-specific exception (the subclass) must come first. If the exceptions are listed with the more-general class first, compilation fails because the more-specific (later) catch block is unreachable. *Java Language Specification* section 14.21 states this, albeit with rather abstruse language. In option C, the suggestion is that adding a catch (Exception) block before the catch (InterruptedException) block would somehow help. But you can immediately see that this will cause, not cure, compilation failure.

Next, you must consider whether an additional catch block is even necessary, or if the right answer might actually be option A, which asserts that the code is good as it stands. It turns out that the definition of the get method of the Future interface is declared to throw checked exceptions. One, as the existing code suggests, is an InterruptedException. It's a matter of general principle that any API in Java that blocks the current thread should break out of the blockage and throw an InterruptedException if the thread receives an interrupt from the Thread.interrupt() method. This design allows code a chance to recover from overly long blockage, or perhaps simply to shut down on request. However, the get method attempts to get the result of the job's execution. Given that the call method of Callable is permitted to throw an exception, it's reasonable to infer that the get method might need to report that exception. Of course, the job itself might have been shut down, and the get method would need an abnormal termination mechanism (that is, another exception) to report that. Because there is more than one possible abnormal termination situation, get actually wraps those in an ExecutionException, and if the call method threw an exception, that exception would be the *cause* of the ExecutionException.

Because of this structure, option A is false (there is a checked exception that must be caught to allow the code to compile) and option D is also false (the checked exception thrown by the get method is not the one declared on the call method implementation). Instead, the "ideal" exception to handle would be ExecutionException, but this isn't offered in the available options. Therefore, option E (catching a simple Exception, but at a valid place in the source code) has to be the correct answer.

There is another observation to be made about this question. The question's construction actually gives away more than a real exam question would typically do. Imagine that you thought the exception that arose from the get method would actually be the one that comes directly from the call method—that is, a MyJobException. That would make both option D and option E correct. But the question calls for only one correct answer. This logic is actually sufficient for you to reject option D. Here, it's a curiosity, but in a real exam question, know that unless there's very specific wording, you are picking correct answers rather than the best answers. And on those very rare occasions where a best answer is called for in one of the programmer certification exams, you can expect very clear wording leading you to understand the criteria that constitute "best." </article>

Simon Roberts joined Sun Microsystems in time to teach Sun's first Java classes in the UK. He created the Sun Certified Java Programmer and Sun Certified Java Developer exams. He wrote several Java certification guides and is currently a freelance educator who teaches at many large companies in Silicon Valley and around the world. He remains involved with Oracle's Java certification projects.

learn more

[Javadoc for the Collection class in Java 8](#)

[Oracle's tutorial on try-with-resources](#)

THE ISTANBUL JUG

In Turkey, Java is one of the most popular programming languages. Istanbul—the huge metropolis connecting Asia and Europe—hosts its own Java user group (JUG). The Istanbul JUG was founded in 2010. Four members set up the leadership board to carry out the primary purpose of the group, which is to contribute to the Java community with the help of talented members. It provides Java news and helps Java developers by blogging; writing ebooks; and holding webinars, workshops, and conferences.

To date, the Istanbul JUG has held 43 meetups and events. The group has also joined the Java Community Process (JCP) program and takes part in Adopt-a-JSR activities. It also contributes to open source projects such as Mongolastic.

So far, the Istanbul JUG has held four Java conferences: Java Day Istanbul 2011, Java Day Istanbul 2013, and Voxxed Days Istanbul 2015 and 2016. Next year's planned Java Day Istanbul 2017, on May 6, is expected to draw 500 attendees. The August 2016 event hosted a hands-on lab based on Java 9 features such as HTTP/2, JShell, and Project Jigsaw. In total, 92 Java developers attended the workshop. They were given programming challenges to learn what's coming in Java 9. As a result of this workshop, the Istanbul JUG received a long list of feedback ideas, helpful for preparing future workshops on JSR 371 (MVC 1.0) and JSR 374 (JSON-P). In October 2016, the Istanbul JUG will hold a workshop about Kubernetes (K8s).

Contact the Istanbul JUG via email or follow it on Twitter and Facebook.

Comments

We welcome your comments, corrections, opinions on topics we've covered, and any other thoughts you feel important to share with us or our readers. Unless you specifically tell us that your correspondence is private, we reserve the right to publish it in our Letters to the Editor section.

Article Proposals

We welcome article proposals on all topics regarding Java and other JVM languages, as well as the JVM itself. We also are interested in proposals for articles on Java utilities (either open source or those bundled with the JDK). Finally, algorithms, unusual but useful programming techniques, and most other topics that hard-core Java programmers would enjoy are of great interest to us, too. Please contact us with your ideas at javamag_us@oracle.com and we'll give you our thoughts on the topic and send you our nifty writer guidelines, which will give you more information on preparing an article.

Customer Service

If you're having trouble with your subscription, please contact the folks at java@halldata.com (phone +1.847.763.9635), who will do whatever they can to help.

Where?

Comments and article proposals should be sent to our editor, **Andrew Binstock**, at javamag_us@oracle.com.

While it will have no influence on our decision whether to publish your article or letter, cookies and edible treats will be gratefully accepted by our staff at *Java Magazine*, Oracle Corporation, 500 Oracle Parkway, MS OPL 3A, Redwood Shores, CA 94065, USA.

☛ Subscription application

☛ Download area for code and other items

☛ *Java Magazine* in Japanese